Hewlett Packard
Enterprise

C/C++ Programmer's Guide for NonStop Systems

Part Number: 862330-004
Published: September 2018
Edition: L15.02 and subsequent L-series, J06.03 and subsequent J-series RVUs, H06.08 and subsequent H-series RVUs, G06.27 and subsequent G-series RVUs

Notices

The information contained herein is subject to change without notice. The only warranties for Hewlett Packard Enterprise products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. Hewlett Packard Enterprise shall not be liable for technical or editorial errors or omissions contained herein.

Confidential computer software. Valid license from Hewlett Packard Enterprise required for possession, use, or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Links to third-party websites take you outside the Hewlett Packard Enterprise website. Hewlett Packard Enterprise has no control over and is not responsible for information outside the Hewlett Packard Enterprise website.

Acknowledgments

Microsoft® and Windows® are trademarks of the Microsoft group of companies.

Adobe® and Acrobat® are trademarks of Adobe Systems Incorporated.

Intel®, Itanium®, Pentium®, and Intel Inside® and the Intel Inside logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

UNIX® is a registered trademark of The Open Group.

Open Software Foundation, OSF, the OSF logo, OSF/1, OSF/Motif, and Motif are trademarks of the Open Software Foundation, Inc.

Warranty

OSF MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THE OSF MATERIAL PROVIDED HEREIN, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

OSF shall not be liable for errors contained herein or for incidental consequential damages in connection with the furnishing, performance, or use of this material.

This software and documentation are based in part on the Fourth Berkeley Software Distribution under license from The Regents of the University of California. OSF acknowledges the following individuals and institutions for their role in its development: Kenneth C.R.C. Arnold, Gregory S. Couch, Conrad C. Huang, Ed James, Symmetric Computer Systems, Robert Elz. © 1980, 1981, 1982, 1983, 1985, 1986, 1987, 1988, 1989 Regents of the University of California. This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit **http://www.openssl.org**.

Export of the information contained in this publication may require authorization from the U.S. Department of Commerce.

Contents

About This Document

This guide describes the implementation of the C and C++ programming languages for HPE NonStop systems. This guide describes these products:

- TNS C compiler in the Guardian and G-series OSS environments.
- TNS C preprocessor (Cprep) and TNS C++ translator (Cfront) in the Guardian and G-series OSS environments.
- TNS/R, TNS/E, and TNS/X native C and C++ compilers in the Guardian and OSS environments.
- TNS/R and TNS/E native C/C++ cross compilers on the PC in NSDEE and ETK.
- TNS/X native C/C++ cross compilers on the PC in NSDEE or from DOS shell. ETK is not supported on TNS/X systems.
- TNS C run-time library in the Guardian and G-series OSS environments.
- TNS/R, TNS/E, and TNS/X native C run-time library in the Guardian and OSS environments.
- TNS C++ run-time library in the Guardian and G-series OSS environments.
- TNS/R, TNS/E, and TNS/X native C++ run-time library in the Guardian and OSS environments.

The term TNS means that a program uses the process, memory, and instruction set architectures of the HPE TNS systems. TNS compilers generate TNS instructions based on complex instruction-set computing (CISC) technology. The Accelerator takes these TNS instructions and generates corresponding instructions based on reduced instruction-set computing (RISC) technology; the Object Code Accelerator takes these TNS instructions and generates corresponding instructions based on Intel Itanium technology. A TNS process is one that runs in TNS mode or accelerated mode.

The term TNS/R native means a program uses the process, memory, and instruction set architectures that are native to RISC processors. The term TNS/E native means a program uses the process, memory, and instruction set architectures that are native to Intel Itanium processors. The term TNS/X native means a program uses the process, memory, and instruction set architectures that are native to Intel x86 processors. The term native mode means a program uses the process, memory, and instruction set architectures that are native to either type of processor.

Native object files consist entirely of RISC, Itanium, or x86 instructions. Unlike TNS processes, native processes do not maintain TNS architecture-specific constructs. Native mode enables you to write programs that are fully optimized for TNS/R, TNS/E, or TNS/X systems such as NonStop servers.

While native mode offers many benefits, you are not required to convert your programs to native mode. The G0x, H0x, J0x, and Lnn release version updates (RVUs) continue to support the TNS compilers and tools. You can continue to create and run TNS and accelerated object code and gain some of the performance benefits provided by code that has been converted to native mode.

When you do mixed language programming, remember these points:

- There are four HPE compilers for the COBOL language, invoked by eight separate commands. All these compilers implement the 1985 standard known as COBOL85 and are released as the product HPE COBOL85 for NonStop Servers:
    - The TNS compiler COBOL85 command in the Guardian environment or the `cobol` command in the G-series Open System Services (OSS) environment
    - The TNS/R native compiler `NMCOBOL` command in the Guardian environment or the `nmcobol` command in the OSS environment

    - The TNS/E native compiler ECOBOL command in the Guardian environment or the `ecobol` command in the OSS environment
    - The TNS/X native compiler XCOBOL command in the Guardian environment or the `xcobol` command in the OSS environment

  An older COBOL language standard, the 1974 version, is no longer supported on HPE NonStop servers. Neither that COBOL 74 product nor its Guardian-environment compiler are included in discussions in this guide unless specifically mentioned.

  The SCREEN COBOL product is also excluded from COBOL discussions in this guide unless specifically mentioned.

  To reduce confusion between the current product name and compiler-specific or platform-specific behaviors, this guide now uses "COBOL" to see the language when a more specific distinction is not needed or is obvious from context. "COBOL85" is used only to refer to the HPE product or to the TNS compiler in the Guardian environment. "TNS COBOL" therefore includes the COBOL85 compiler, the OSS `cobol` compiler, or the TNS version of the COBOL run-time library, while "native COBOL" can refer to either the TNS/R, TNS/E or TNS/X compiler.

- Similarly, there are four compilers in the Guardian environment for dialects of the Transaction Application Language (TAL):
    - The TAL compiler, which produces TNS code for the original language dialect of TAL.
    - The PTAL compiler, which produces TNS/R native code for the second, portable dialect of TAL (pTAL)
    - The EPTAL compiler, which produces TNS/E native code for the dialect of pTAL available for execution on H-series RVUs.
    - The XPTAL compiler, which produces TNS/X native code for the dialect of pTAL available for execution on L-series RVUs.

  Unless otherwise specified in text, references to pTAL apply to code produced by either the PTAL, EPTAL, or XPTAL compilers.

Supported Release Version Updates (RVUs)

This manual supports L15.02 and all subsequent L-series, J06.03 and all subsequent J-series RVUs, H06.08 and all subsequent H-series RVUs, G06.27 and all subsequent G-series RVUs until otherwise indicated in a replacement publication.

Intended Audience

This guide is intended for systems and applications programmers familiar with the C and C++ programming languages, HPE NonStop servers, and the HPE NonStop OS. This manual does not describe the C/C++ standards, but does describe HPE extensions and enhancements to the standards, in addition to implementation-defined behavior.

New and Changed Information

Changes for September 2018 (862330-004)

Updated the following:

- **C++ Run-Time Library and Standard C++ Library**
- **c11 support**
- **Usage Guidelines**

Added the compiler pragma, **THREAD**.

Added **Atomic Intrinsic Functions**.

Changes for April 2018 (862330-003)

- Added new section **PROFILING/NOPROFLING** on page 286.
- Added new section **TARGET** on page 320.
- Updated the section **DEBUG_USES_LINE_DIRECTIVES** on page 219.
- Updated the following sections for pragma PROFILING/NOPROFILING:
  - **CODECOV** on page 214
  - **PROFGEN** on page 287
  - **PROFUSE** on page 288
- Updated the description for TARGET pragma in the **Table 27: Compiler Pragma Descriptions** on page 193.

Changes for March 2017 (862330-002)

- Added information for C11 support and C++11 support in **c11 support** and **c++11 Support** respectively.
- Added information for TNS/X Native C and C++ Language System in **C++ Run-Time Library and Standard C++ Library**, **C++ Header Files**, **TNS/X Native Linker (xld Utility)**, **TNS/X Native C++ Compiler**, and **Features of TNS/X Native C and C++**.
- Added VERSION4 information for Using the Standard C++ Library in **Using the Standard C++ Library**.
- Added VERSION4 information for Contents of the Standard C++ Library in **VERSION4**, **Installation Notes for VERSION4**, and **Using Header Files With VERSION4**.
- Added information for Compiler Pragmas in **BOOST**, **c99**, **c11**, **TEMPEXTENT**, and **VERSION4**.
- Added information for Compiling and Linking TNS/R Native C and C++ Programs in **Using Compiler Pragmas IEEE_Float**
- Added information for Compiling and Linking TNS/X Native C and C++ Programs in **Compiling and Linking TNS/X Native C and C++ Programs**
- Added information for Determining Which DLLs are Required in **Determining Which DLLs are Required**.
- Added information for Using the Native C/C++ Cross Compiler on the PC in **C/C++ PC Cross Compiler**

- Added information for C and C++ Extensions in **GCC compatible language extensions**.
- Added information for Boost Libraries Support in **Boost Libraries Support** on page 110, **Compiling with Boost**, and **BOOST** on page 208.

Changes for June 2016 (862330-001)

Added information for C11 support in **c11 support** on page 605 and **TNS/X Native C Compiler** on page 42.

Added information about SSE and other intrinsic functions in **SSE Intrinsic Functions** on page 401 and **Other intrinsic functions** on page 405.

Added information for Boost libraries in **Boost Libraries Support** on page 110, **_typeof** on page 70 and **BOOST** on page 208.

Changes for August 2015 (429301-019)

Added information about AES intrinsic functions in **Specifying Header Files** on page 399 and **AES Intrinsic Functions** on page 400.

Related Information

In addition to the HPE C and C++ documentation set, these tables list other manuals that can be useful to a C or C++ programmer.

Table 1: Program Development Manuals and Online Help

Manual Title	Description
Accelerator Manual	Describes how to use the Accelerator to optimize TNS object code for the TNS/R execution environment.
Binder Manual	Describes the Binder program, an interactive linker that enables you to examine, modify, and combine object files and to generate load maps and cross-reference listings.
Common Run-Time Environment (CRE) Programmer's Guide	Describes the common run-time environment (CRE) and how to write and run mixed-language programs.
`eld` *and* `xld` *Manual*	Describes how to use the TNS/E and TNS/X native PIC (Position-Independent Code) linkers `eld` and `xld`.
`enoft` *Manual*	Describes how to use the TNS/E native object file tool `enoft`.
NonStop Development Environment for Eclipse (NSDEE) online help	Describes how to use the NonStop Development Environment for Eclipse (NSDEE), a PC-based cross-development environment for building native C/C++ and native COBOL applications targeted at NonStop systems.

Table Continued

Manual Title	Description
HPE Enterprise Toolkit—NonStop Edition (ETK) online help	Describes how to use ETK, a PC-based cross-development environment for building native C/C++ applications targeted at NonStop systems. With the optional cross compilers, ETK allows you to build pTAL and native COBOL applications for NonStop servers. ETK is compatible with both NonStop SQL/MP and NonStop SQL/MX. ETK requires Visual Studio.net. See **Using the Native C/C++ Cross Compiler on the PC** on page 419.
H-Series Application Migration Guide	Describes the TNS/E native development and execution environments and describes how to migrate existing TNS/R applications to TNS/E native applications.
Inspect Manual	Describes the Inspect program, an interactive source-level or machine-level debugger that enables you to interrupt and resume TNS/R program execution, and display and modify variables.
L-Series Application Migration Guide	Describes the TNS/X native development and execution environments and describes how to migrate existing TNS/E applications to TNS/X native applications.
`ld` *Manual*	Describes how to use the TNS/R native PIC linker `ld`.
Native Inspect Manual	Describes the H-series Native Inspect program, an interactive source-level or machine-level debugger that enables you to interrupt and resume TNS/E program execution, and display and modify variables.
`nld` *Manual*	Describes how to use the native non-PIC linker nld.
`noft` *Manual*	Describes how to use the TNS/R native object file tool noft.
Object Code Accelerator (OCA) Manual	Describes how to use the Object Code Accelerator to optimize TNS object code for the TNS/E execution environment.
`rld` *Manual*	Describes how to use the native PIC loader `rld`.
SQL/MP Programming Manual for C	Describes how to use NonStop SQL/MP with the TNS, TNS/R, and TNS/E native C compilers.
SQL/MX Programming Manual for C and COBOL	Describes how to use NonStop SQL/MX with the native C and C++ compilers in the OSS environment only.
Standard C++ Library Class Reference; Standard C++ Library User Guide and Tutorial	Describe how to use the VERSION2 Standard C++ Library (also see **Using the Standard C++ Library** on page 88).
Standard Compliant C++ Library Reference	Describes the functions and elements in the VERSION3 Standard C++ Library (also see **Using the Standard C++ Library** on page 88).

Table Continued

Manual Title	Description
TNS/R Native Application Migration Guide	Describes the TNS/R native development and execution environments and describes how to migrate existing TNS applications to TNS/R native applications.
Tools/h++ Class Reference; Tools.h++ Manual; Tools.h++ 7.0 User Guide	Describes how to use the release 6.1 and 7.0 Tools.h++ class libraries (also see **Accessing Middleware Using HPE C and C++ for NonStop Systems** on page 104).
Code Profiling Utilities Manual	Explains how to use the HPE Code Profiling Utilities to perform profile-guided optimization and to generate code coverage reports.
Visual Inspect online help	Describes how to use Visual Inspect, a graphical symbolic debugging product that provides powerful data display, application navigation, and multi-program support capabilities. Visual Inspect consists of a Windows client and a NonStop host-based server.
`xnoft` Manual	Describes how to use the TNS/X native object file tool `xnoft`.

Table 2: Guardian Environment Manuals

Manual Title	Description
Guardian Programmer's Guide	Describes how to use Guardian procedure calls to access operating system services from an application.
Guardian Procedure Calls Reference Manual	Describes the syntax for Guardian procedure calls.
Guardian Procedure Errors and Messages Manual	Describes error codes, error lists, system messages, and trap numbers for Guardian system procedures.
Guardian Application Conversion Guide	Describes how to convert C, COBOL, Pascal, TAL, and TACL applications to use the extended features of the HPE NonStop OS.

Table 3: Open System Services (OSS) Environment Manuals

Manual Title	Descriptions
Open System Services Library Calls Reference Manual	Describes the syntax and semantics of the native C run-time library in the OSS environment.
Open System Services Programmer's Guide	Describes how to use the OSS application programming interface to the operating system.

Table Continued

Manual Title	Descriptions
Open System Services Shell and Utilities Reference Manual	Describes the syntax and semantics for using the OSS shell and utilities.
Open System Services System Calls Reference Manual	Describes the syntax and programming considerations for using OSS system calls.

Publishing History

Part Number	Product Version	Publication Date
862330-004	G07, H01, and L01	September 2018
862330-003	G07, H01, and L01	April 2018
862330-002	G07, H01, and L01	March 2017
862330-001	G07, H01, and L01	June 2016
429301-019	G07, H01, and L01	August 2015

Introduction to HPE C and C++ for NonStop Systems

TNS C Language System

The HPE TNS C language system for NonStop servers generates TNS C programs for the Guardian and G-series OSS environments. The C preprocessor, C compiler, C run-time library, and Binder comprises the TNS C language system. Optional components of this system include the Accelerator, the TNS Object Code Accelerator (OCA), the Inspect or Visual Inspect symbolic debuggers, and the NonStop SQL/MP compiler.

HPE C for NonStop systems conforms to the C language standard as set forth in ISO/IEC 9899:1990, Programming Languages–C. (This is technically identical to ANSI C X3.159-1989, Programming Language C.) HPE C also includes several extensions to ISO/ANSI standard C that makes C an effective language for writing applications that run on NonStop systems. For more details, see **C and C++ Extensions** on page 53.

C Preprocessor

The C preprocessor, Cprep, manipulates the text of a source file as the first phase of compilation. Run Cprep as an independent process in NonStop environment. The C preprocessor is embedded in the C compiler that runs in the Guardian environment. The C compiler that runs in the OSS environment does not contain the C preprocessor; Cprep must be run separately. (The OSS TNS `c89` utility runs the C preprocessor by default.)

C Compiler

The C compiler then translates C source text into machine language code. If there are mistakes in your program, error messages are generated; otherwise, object code is generated.

C Run-Time Library

The C run-time library contains the functions defined in the ISO/ANSI C standard in addition to several functions that are extensions to the standard. Macros that are not in the library are expanded during the preprocessing phase of the compiler.

The Guardian C run-time library and the OSS C run-time library share the same set of header files. Header files contain function prototypes and the variable and type declarations used by the functions.

For the complete semantics and syntax of the C run-time library in the Guardian TNS C environment, see the *Guardian TNS C Library Calls Reference Manual*. Use of TNS C in the G-series OSS environment was discouraged after the D40 RVU and the OSS TNS library calls are no longer completely documented. TNS C is not available in the TNS/E and TNS/X OSS environment.

Binder

The Binder collects and links object files generated by the TNS C compiler and produces an executable TNS object file (a program file).

Accelerator

The Accelerator enables you to increase the performance of TNS programs that runs on TNS/R system. The Accelerator optimizes TNS program files (compiled and bound object files) to take advantage of the TNS/R architecture.

Object Code Accelerator (OCA)

The Object Code Accelerator (OCA) enables you to increase the performance of TNS programs that runs on TNS/E systems, and the Object Code Accelerator (OCAX) for TNS/X that runs on TNS/X systems. The OCA optimizes TNS program files (compiled and bound object files) to take advantage of the TNS/E and TNS/X architecture.

Inspect Symbolic Debugger

The Inspect symbolic debugger is an interactive tool that enables you to identify and correct programming errors in programs.You can use the Inspect debugger to:

- Step through code
- Set breakpoints
- Display source
- Display variables

For more details on TNS program debugging capabilities, see the *Inspect Manual* and the G-series *Debug Manual*.

Visual Inspect Symbolic Debugger

Visual Inspect is an optional PC-based (GUI) symbolic debugger designed to work in complex distributed environments. Visual Inspect:

- Supports application debugging in either the development or production environment
- Uses program visualization, direct manipulation, and other techniques to improve productivity for both new and sophisticated users
- Provides source-level debugging for servers executing in NonStop environment; provides additional application navigation features that allow a higher level of abstraction
- Supports both TNS and native machine architectures and compilers (that is, C, C++, COBOL85, NMCOBOL, ECOBOL, EpTAL, TAL, pTAL, D-series Pascal, and FORTRAN), in the Guardian and OSS executing environments
- Is also available as a stand-alone product

For more details about enabling and using Visual Inspect, see the Visual Inspect online help. See also the descriptions of pragmas **INSPECT** on page 256 and **SYMBOLS** on page 315.

NOTE:

Visual Inspect is not supported on TNS/X systems. On both TNS/E and TNS/X systems, if you want GUI you must use NSDEE as the debugger on Windows environment. For command line debugger, you can use Native Inspect.

NonStop SQL/MP Compiler

The NonStop SQL/MP compiler processes embedded SQL statements from C source programs and generates the correct NonStop SQL/MP database calls.

TNS C++ Language System

The TNS C++ language system includes the TNS C language system and the HPE C++ translator, Cfront. Cfront supports the C++ language features provided in the UNIX System Labs (USL) C++ translator, Cfront, version 3.0. These features are described in *The Annotated C++ Reference Manual* (ANSI Base Document). The HPE C++ run-time library provides the functions needed for initialization, termination, and memory management of C++ programs and the `iostream` class library.

HPE Cfront accepts C++ source code accepted by USL Cfront, version 3.0. In addition, Cfront provides support for:

- C compiler pragmas
- Large-memory model
- 32-bit data model
- Data type `signed char`
- Data type `long long`
- Source level debugging with the Inspect debugger

In the Guardian environment, support is also provided for:

- Mixed language programming using interface declarations
- Completion codes for error detection

Exception Handling Is Not Supported

Exception handling is described in *The Annotated C++ Reference Manual*. However, exception handling is not implemented in either USL Cfront or HPE Cfront.

Compilation Steps for C++ Code

To compile C++ code in the Guardian and OSS environments, use these steps:

1. Run the Cprep preprocessor to provide macro and file expansion. The C preprocessor produces C++ source code for input to Cfront.
2. Run the Cfront translator to translate the C++ source code and produce C source code for input to the C compiler.
3. Run the C compiler to produce C object code that, when bound with the C++ run-time library and the C run-time library, produces an executable C++ program.
4. You can optionally process the executable C++ program with the Accelerator or OCA.

In the Guardian environment, you can use a TACL macro to run the compilation system components. In the OSS environment, the G-series TNS `c89` utility (`/nonnative/bin/c89`) provides an interface to the C++ compilation system components. For more details, see **Compiling, Binding, and Accelerating TNS C++ Programs** on page 351.

TNS/R Native C and C++ Language System

The TNS/R native C and C++ language system generates TNS/R native C and C++ programs for the Guardian and OSS environments.

Components of this language system:

- **TNS/R Native C Compiler** on page 28
- **TNS/R Native C++ Compiler** on page 29
- **TNS/R Native C Run-Time Library** on page 29
- **C++ Run-Time Library and Standard C++ Library** on page 29
- **TNS/R Native Linkers (nld and ld Utilities)** on page 30
- Optional components:
  - **Inspect Symbolic Debugger** on page 31
  - **Visual Inspect Symbolic Debugger** on page 31
  - **NonStop SQL/MP Compiler and NonStop SQL/MX Compiler** on page 32
  - **TNS/R Native C and C++ Migration Tool** on page 32

TNS/R Native C Compiler

The TNS/R native C compiler accepts C language source files that comply with the ISO/ANSI C Standard (ISO/IEC 9899:1990, Programming Languages–C or ANSI C X3.159-1989, Programming Language C) or Common-Usage C (sometimes called Kernighan and Ritchie C, or K&R C). The native C compiler also accepts HPE NonStop extensions that support the native architecture.

The native C compiler can be run in the Guardian and OSS environments or on a PC:

- In the Guardian environment, use the NMC command to run the native compiler. The NMC command line syntax is similar to that of the C command for running the TNS C compiler. For syntax information, see **Compiling a C Module** on page 338.
- In the OSS environment, use the native `c89` utility to run the TNS/R native C compiler. The native `c89` utility syntax is similar to that of the TNS `c89` utility. For syntax information, see the `c89(1)` reference page either online or in the *Open System Services Shell and Utilities Reference Manual*. The *Open System Services Programmer's Guide* provides guidance on the use of C in the OSS environment.
- On a PC running the Windows operating system, use the NonStop Development Environment for Eclipse (NSDEE) or ETK to compile C code. You can also use the command-line cross compiler (named `c89`) outside ETK. For more details, see the online help in NSDEE or ETK, or the file "Using the Command-Line Cross Compilers" installed with ETK compiler package.

  NOTE: For more information about ETK, see **Using the Native C/C++ Cross Compiler on the PC** on page 419. ETK is not supported on TNS/X systems. ETK is supported only on TNS/R and TNS/E systems. It is not supported on TNS/X systems.

The native C compiler supports programs that define the size of pointers and type `int` as 32 bits (programs compiled with the pragma `WIDE`). Existing TNS C language programs that define pointers or type `int` as 16 bits must be changed. Few other C language source code changes are required to use the native C compiler.

TNS/R Native C++ Compiler

There are three versions or dialects of the native C++ compiler; all the three versions accept C++ language source files and support HPE language extensions. However, the three versions support different standards as described in the descriptions of pragmas **VERSION1** on page 322, **VERSION2** on page 324, and **VERSION3** on page 326.

The native C++ compiler can be run in the Guardian and OSS environments, and on a PC using NSDEE or ETK:

- In the Guardian environment, use the NMCPLUS command to run the native C++ compiler. NMCPLUS command syntax is similar to that of the C command for running the TNS C compiler and to that of the Cfront translator. For syntax information, see **Compiling a C Module** on page 338.
- In the OSS environment, use the native `c89` utility to run the native C++ compiler. Native `c89` syntax is similar to that of the TNS `c89` utility. For syntax information, see the `c89(1)` reference page either online or in the *Open System Services Shell and Utilities Reference Manual*.
- On a PC running the Windows operating system, use the NonStop Development Environment for Eclipse (NSDEE) or ETK to compile C code. You can also use the command-line cross compiler (named `c89`) outside of NSDEE or ETK. For more details, see the online help in NSDEE or ETK, or the file “Using the Command-Line Cross Compilers” installed with ETK compiler package.

NOTE: For more information about NSDEE and ETK, see **Using the Native C/C++ Cross Compiler on the PC** on page 419. ETK is not supported on TNS/X systems.

The native C++ compiler supports programs that define the size of data type `int` as 32 bits (programs compiled with the pragma `WIDE`). Existing TNS C++ language programs that define the type `int` as 16 bits must be changed. Few other C++ language source code changes are required to use the native C++ compiler.

The native C++ compiler provides a more powerful and simplified development environment than TNS Cfront. For example, to create an accelerated executable TNS C++ program, you must perform five steps (running Cprep, Cfront, the TNS C compiler, Binder, and the Accelerator). In comparison, to create an executable native C++ program, you run only the native C++ compiler and a TNS/R native linker.

TNS/R Native C Run-Time Library

The native C run-time library provides functions conforming to the ISO/ANSI C Standard. It also contains functions conforming to the X/OPEN UNIX 95 specification and HPE extensions to these standards.

The native C run-time library supports Guardian and OSS processes. The native C run-time library is nearly identical for the Guardian and OSS environments and therefore increases the interoperability between environments. For more details on interoperability, see the *Open System Services Programmer’s Guide*.

The native C run-time library provides locale-sensitive functions and algorithmic code-set converters for use in internationalized OSS applications. For more details, see the *Software Internationalization Guide*.

C++ Run-Time Library and Standard C++ Library

The C++ run-time library and the Standard C++ Library are available to every C++ program. However, there are three versions of the libraries, as listed below and in the table **Versions of the Standard C++ and C++ Run-Time Libraries** . Specifying a version establishes a context that includes the dialect of the native C++ compiler, the run-time libraries available to you, and the libraries that are automatically linked when you compile a executable object file.

VERSION1 C++ Library

For C++ **VERSION1** on page 322 (the default version before the G06.20 RVU), these libraries are available:

- The HPE NonStop C run-time library (file ZCRTLSRL)
- The HPE NonStop C++ run-time library (product T9227, Guardian version in file ZCPLGSRL; OSS version in file ZCPLOSRL)
- Tools.h++ version 6.1 (Guardian version in file ZTLHGSRL; OSS version in file ZTLHOSRL)

VERSION2 Standard C++ Library

For C++ **VERSION2** on page 324, these libraries are available:

- The HPE NonStop C run-time library (file ZCRTLSRL)
- The HPE NonStop C++ run-time library (product T0179, file ZCPLSRL)
- The draft Standard C++ Library from Rogue Wave (product T5895, file ZRWSLSRL)
- Tools.h++ version 7.0 (product T8473, file ZTLHSRL)

VERSION3 Standard C++ Library ISO/IEC

For C++ VERSION3 (the default version beginning with the G06.20 RVU), these libraries are available:

- The HPE NonStop C run-time library (file ZCRTLSRL)
- The **VERSION3** on page 326 library (product T2767, file ZSTLSRL), which includes:
    - The HPE NonStop C++ run-time library
    - The ratified Standard C++ Library ISO/IEC from Dinkumware, based on the standard document: ISO/IEC 14882:1998(E).

C++ Header Files Combined at G06.20

At the G06.20 RVU, the C++ header files for all three versions were combined into one product number, T2824. Built-in checking enforces the use of the correct header files with each version. This change did not affect how you use the header files; the change is transparent to the user.

Tools.h++ version 7 headers remain in product T5894.

TNS/R Native Linkers (nld and ld Utilities)

The `nld` native linker links one or more TNS/R native object files (files generated by the native compilers or `nld`) to produce either an executable (a loadfile) or relinkable native object file (linkfile). `nld` can also modify process attributes, such as HIGHPIN, of executable native object files and remove nonessential information from native object files. `nld` cannot process position-independent code (PIC) files.

The `ld` native linker, introduced at G06.20, performs the same basic functions as `nld,` but processes PIC files, the format necessary when using dynamic-link libraries (DLLs). If you are using DLLs, you must use `ld` to link your linkfiles.

Unlike Binder, TNS/R native linker cannot replace individual procedures and data blocks in an object file or build an object file from individual procedures and data blocks. TNS/R native linker operate on procedures and data blocks, but only in terms of an entire object file.

The TNS/R native linker runs in the NonStop environments in addition to NSDEE and ETK on the PC. With both linkers, you can specify the target platform as NonStop. Command syntax and capabilities are nearly identical in each environment. To display the syntax, enter **`nld`** or **`ld`** at the command prompt.

For more details, see the:

- *DLL Programmer's Guide for TNS/R Systems*
- *ld Manual*
- *nld Manual*
- *noft Manual*
- *rld Manual*

Inspect Symbolic Debugger

The Inspect symbolic debugger is an interactive tool that enables you to identify and correct programming errors in programs.

You can use Inspect to:

- Step through code
- Set breakpoints
- Display source
- Display variables

For more details on debugging capabilities, see the *Inspect Manual* and the G-series *Debug Manual*.

Visual Inspect Symbolic Debugger

Visual Inspect is an optional PC-based (GUI) symbolic debugger designed to work in complex distributed environments. Visual Inspect:

- Supports application debugging in either the development or production environment
- Uses program visualization, direct manipulation, and other techniques to improve productivity for both the new and sophisticated users
- Provides source-level debugging for servers executing in NonStop environment; provides additional application navigation features that allow a higher level of abstraction
- Supports both TNS and native machine architectures and compilers (that is, C, C++, COBOL85, NMCOBOL, ECOBOL, EpTAL, TAL, pTAL, D-series Pascal, and FORTRAN), in the Guardian and OSS executing environments
- Is available also as a stand-alone product

For more details about enabling and using Visual Inspect, see the Visual Inspect online help. See also the descriptions of pragmas **INSPECT** on page 256 and **SYMBOLS** on page 315.

NOTE:

Visual Inspect debugger is not supported on TNS/X systems.

TNS/R Native Object File Tool (noft Utility)

The native object file tool (the `noft` utility) reads and displays information about TNS/R native object files. You can use `noft` to:

- determine the optimization level of procedures
- display object code with corresponding source code
- list object file attributes
- list unresolved references

`noft` runs in the Guardian and OSS environments. The `noft` syntax and capabilities are nearly identical in each environment. For more details, see the *noft Manual.*

Native object files are in Executable and Linking Format (ELF), a standard format used for object files, with HPE extensions. For more details on native object files, see the *nld Manual* and *ld Manual.*

A native object file is either a linkfile or a loadfile, but not both. The native compilers produce only linkfiles, while the TNS/R native linker utility can produce either linkfiles or loadfiles.

Linkfiles can be used again as input to TNS/R native linker. Loadfiles can be used as input to TNS/R native linker only for modifying loadfile attributes. Both linkfiles and loadfiles can be used as `noft` input.

	Can Be Linked to Produce a Loadfile	**Can Be Executed**
Linkfiles	Yes	No
Loadfiles	No	Yes

NonStop SQL/MP Compiler and NonStop SQL/MX Compiler

The NonStop SQL/MP compiler processes embedded SQL statements from C source programs and generates the correct NonStop SQL/MP database calls. For more details see the SQL/MP Programming Manual for C.

The NonStop SQL/MX compiler, available on the OSS platform, processes embedded SQL statements from TNS/R native C or C++ source programs and generates the correct NonStop SQL/MX database calls. For more details see the *SQL/MX Programming Manual for C and COBOL*.

TNS/R Native C and C++ Migration Tool

The native C and C++ migration tool, NMCMT, scans source files and produces a diagnostic listing. The listing identifies most C and C++ language source code changes required to migrate from TNS C (D20 or later product versions) to native C or C++.

The native mode migration tool is available in both the Guardian and OSS environments, and it is integrated into ETK on the PC.

Features of TNS/R Native C and C++

Over the years, TNS/R native C and C++ have evolved as the languages themselves were changing.

- At the D40 release, the TNS/R native C/C++ compiler accepted the language as defined by *The Annotated C++ Reference Manual* by Ellis and Stroustrup (excluding support for the exception handling).
- At D45, some features were updated to match the language specification in the 1996 *X3J16/WG21 Working Paper*. VERSION2 of the native C++ compiler is based on the 1996 standard. The native C++ compiler introduced support for exceptions and a number of features that are not in the *The Annotated C++ Reference Manual* but are new in the working paper. A summary of the major features added at D45 appears in the description of pragma **VERSION2** on page 324. A list of specific features accepted from the working paper appears in **Features and Keywords of Version 2 Native C++** on page 565.
- At G06.20, a new version or dialect of the native C++ compiler was introduced. VERSION3 of the compiler is based on the 1998 standard, listed in the references that follow. VERSION3 is the default library used by the native C++ compiler.

This manual is not intended to be a reference manual for ANSI C or for C++. For a complete description of ANSI C, see ANSI X3.159. For a complete description of C++, see ANSI X3J16/96-0225 (VERSION2) and ISO/IEC 14882:1998(E) (VERSION3).

Useful references on C and C++ include:

- *ANSI C*, American National Standards Institute. ANSI X3.159-1989.
- Ellis, Margaret A. and Stroustrup, Bjarne. *The Annotated C++ Reference Manual*. Addison Wesley, 1990.
- ISO/IEC. *Programming Languages – C*. International Standard ISO/IEC 9899. First edition 1990-12-15.
- *Working Paper for Draft Proposed International Standard for Information Systems–Programming Language C++*. X3, Information Processing Systems. 2 Dec 1996. X3J16/96-0225 WG21/N1043 (the standard on which the NonStop VERSION2 of the Standard C++ Library is based).
- International Standard ISO/IEC 14882:1998(E) Programming Languages -- C++ (the 1998 standard on which the NonStop VERSION3 of the Standard C++ Library is based).
- International Standard ISO/IEC 14882:2003(E) Programming Languages -- C++ (a newer standard)

HPE includes several extensions to ISO/ANSI standard C that make C an effective language for writing the applications that execute under the HPE NonStop OS. Some of these features are:

- Access to several types of physical files

  There is an extensive set of I/O library routines that enable you to access many different types of physical files, including:

  - C disk files, which are odd-unstructured files and have a file code of 180
  - EDIT disk files, which have a file code of 101
  - Processes
  - Terminals
  - $RECEIVE

- Two file-reference models

There are two sets of input and output routines; each set has its own method of tracking, maintaining, and referring to a file. These methods are called file-reference models, they are:

  - The ANSI model, which uses FILE pointers to identify the files.
  - The alternate or UNIX-style model, which uses file descriptors to identify the files.

    With two file-reference models available, select the model whose I/O services best suit the needs of application. For more details, see **Using the C Run-Time Library** on page 78.

- Access to Guardian system procedures

  You can call procedures in the Guardian system library using the `cextdecs` header file in the Guardian environment or the `cextdecs.h` header file in the OSS or PC environments.

- Access to Open System Services system calls and library calls

  Call routines that are part of the Open System Services library.

- Access to procedures written in other languages

  TNS programs can call procedures written in C, C++, TNS COBOL, FORTRAN, D-series Pascal, and TAL. Native programs can call procedures written in C, C++, native COBOL, and pTAL.

- Access to a NonStop SQL/MP database or a NonStop SQL/MX database

  TNS/R native C or C++ program can interface to a NonStop SQL/MP or NonStop SQL/MX database using embedded SQL.

- Fault-tolerant programs

  Write fault-tolerant process pairs using the active backup programming model.

TNS/E Native C and C++ Language System

The TNS/E native C and C++ language system generates TNS/R and TNS/E native C and C++ programs for the Guardian and OSS environments.

Components of this language system:

- **TNS/E Native C Compiler** on page 35
- **TNS/E Native C++ Compiler** on page 36
- **TNS/E Native C Run-Time Library** on page 36
- **C++ Run-Time Library and Standard C++ Library** on page 36
- **TNS/E Native Linker (eld Utility)** on page 37
- Optional components:
  - **Native Inspect Symbolic Debugger** on page 38
  - **Visual Inspect Symbolic Debugger** on page 38

- ◦ **NonStop SQL/MP Compiler and NonStop SQL/MX Compiler** on page 39
- ◦ **TNS/E Native C and C++ Migration Tool** on page 39

TNS/E Native C Compiler

The TNS/E native C compiler accepts:

- C language source files that comply with the ISO/ANSI C Standard (ISO/IEC 9899:1990, Programming Languages–C or ANSI C X3.159-1989, Programming Language C) or Common-Usage C (sometimes called Kernighan and Ritchie C, or K&R C).
- For H06.08 and later H-series RVUs and J06.03 and later J-series RVUs, selected features from the 1999 update to this standard (ISO/IEC 9899:1999) (see **c99 Selected Features (C99LITE)** on page 598, for a summary of these features).

  NOTE: To use the supported features from the 1999 standard, you must specify the `C99LITE` pragma on the compiler command line (Guardian) or the `-Wc99lite` option on the `c89` command (OSS, Windows).

For H06.21 and later H-series RVUs and J06.10 and later J-series RVUs, for TNS/E native applications that use IEEE floating point, all features from the 1999 update to this standard (ISO/IEC 9899:1999). Complex types and several new math functions and macros are not supported for Tandem floating point format, but all other c99 features are available when compiled using the Tandem floating point format option. For more information about this support, see **c99 Full Support** on page 602.

NOTE: To use these features:

- When compiling on Guardian, use the C99 option of the CCOMP command to enable compiling to the 1999 standard (for example `CCOMP/ in filec/;c99`. The default is to compile according to the 1989 standard.
- When compiling on OSS and Windows, use `c89` to compile using the 1989 standard or use `c99` to compile using the 1999 standard.

HPE NonStop extensions that support the native architecture.

The TNS/E native C compiler can be run in the Guardian and OSS environments, and on a PC using the NonStop Development Environment for Eclipse (NSDEE) or ETK:

- In the Guardian environment, use the CCOMP command to run the TNS/E native compiler. The CCOMP command-line syntax is similar to that of the NMC command for running the TNS/R C compiler. For syntax information, see **Compiling a C Module** on page 338.
- In the OSS environment, use the native `c89` utility or the `c99` utility to run the TNS/E native C compiler. For syntax information, see the `c89(1)` or the `c99(1)` reference page either online or in the *Open System Services Shell and Utilities Reference Manual*. The *Open System Services Programmer's Guide* provides guidance on the use of C in the OSS environment.
- On a PC running the Windows operating system, use NSDEE or ETK to compile C code. You can also use the command-line cross compiler (named `c89`) outside NSDEE and ETK. For more details, see the NSDEE online help, the ETK online help, or the file "Using the Command-Line Cross Compilers on

Windows” installed with ETK compiler package. ETK is supported only on TNS/R and TNS/E systems. It is not supported on TNS/X systems.

The TNS/E native C compiler supports programs that define the size of pointers and type `int` as 32 bits (programs compiled with the pragma `WIDE`). Existing TNS C language programs that define pointers or type `int` as 16 bits must be changed. Few other C language source code changes are required to use the native C compiler.

TNS/E Native C++ Compiler

TNS/E native C++ compiler accepts one of two dialects of the C++ language; both versions accept C++ language source files and support HPE language extensions. However, the versions support different standards as described in the descriptions of pragmas **VERSION2** on page 324 and **VERSION3** on page 326.

The TNS/E native C++ compiler can be run in the Guardian and OSS environments, and on a PC using NSDEE or ETK:

- In the Guardian environment, use the CPPCOMP command to run the native C++ compiler. CPPCOMP command syntax is similar to that of the NMCPLUS command for running the TNS/R C++ compiler. For syntax information, see **Compiling a C Module** on page 338.
- In the OSS environment, use the native `c89` utility or the `c99` utility to run the TNS/E native C compiler. For syntax information, see the `c89(1)` or the `c99(1)` reference page either online or in the *Open System Services Shell and Utilities Reference Manual*. The *Open System Services Programmer’s Guide* provides guidance on the use of C in the OSS environment.
- On a PC running the Windows operating system, use NSDEE or ETK to compile C code. You can also use the command-line cross compiler (named `c89` or `c99`) outside NSDEE and ETK. For more details, see the NSDEE online help, the ETK online help, or the file “Using the Command-Line Cross Compilers on Windows” installed with ETK compiler package.

The TNS/E native C++ compiler supports programs that define the size of data type `int` as 32 bits (programs compiled with the pragma `WIDE`). Existing TNS C++ language programs that define the type `int` as 16 bits must be changed. Few other C++ language source code changes are required to use the native C++ compiler.

The native C++ compiler provides a powerful and simplified development environment. For example, to create an executable native C++ program, you run only the native C++ compiler and the `eld` native linker.

TNS/E Native C Run-Time Library

The native C run-time library provides functions conforming to the ISO/ANSI C Standard. It also contains functions conforming to the X/OPEN UNIX 95 specification and HPE extensions to these standards.

The native C run-time library supports Guardian and OSS processes. The native C run-time library is nearly identical for the Guardian and OSS environments and therefore increases the interoperability between environments. For more details on interoperability, see the *Open System Services Programmer’s Guide*.

The native C run-time library provides locale-sensitive functions and algorithmic code-set converters for use in internationalized OSS applications. For more details, see the *Software Internationalization Guide*.

C++ Run-Time Library and Standard C++ Library

The C++ run-time library and the Standard C++ Library are available to every C++ program. However, there are two versions of the libraries, as listed in this subsection and in Table 5-1 on page 5-2. Specifying a version establishes a context that includes the dialect of the native C++ compiler, the run-time libraries available, and the libraries that are automatically linked when compile a executable object file.

VERSION2 Standard C++ Library

For C++ **VERSION2** on page 324, these libraries are available:

- The HPE NonStop C run-time library (file ZCRTLDLL)
- The HPE NonStop C++ common run-time library (product T2831, file ZCPPCDLL)
- The VERSION2-specific draft Standard C++ Library from Rogue Wave (product T2832, file ZCPP2DLL)
- Tools.h++ version 7.0 (product T2835, file ZTLH7DLL)

VERSION3 Standard C++ Library ISO/IEC

For C++ **VERSION3** on page 326 (the default version), these libraries are available:

- The HPE NonStop C run-time library (file ZCRTLDLL)
- The VERSION3 library, which includes:
  - The HPE NonStop C++ common run-time library (product T2831, file ZCPPCDLL)
  - The VERSION3-specific ratified Standard C++ Library ISO/IEC from Dinkumware, based on the standard document: ISO/IEC 14882:1998(E) (product T2833, file ZCPP3DLL).

C++ Header Files

The C++ header files for the three versions are combined in product T2830. Built-in checking enforces the use of the correct header files with each version.

Tools.h++ version 7 header files are found in product T2834.

TNS/E Native Linker (eld Utility)

The `eld` native linker links one or more TNS/E native object files (files generated by the native compilers or `eld`) to produce either an executable (a loadfile) or relinkable native object file (linkfile). `eld` can also modify process attributes, such as HIGHPIN, of executable native object files and remove nonessential information from native object files.

The `eld` native linker processes PIC (position-independent code), the format necessary when using dynamic-link libraries (DLLs).

`eld` cannot replace individual procedures and data blocks in an object file or build an object file from individual procedures and data blocks. `eld` operates on procedures and data blocks, but only in terms of an entire object file.

`eld` runs in the Guardian and OSS environments in addition to ETK on the PC. You can specify the target platform as NonStop. Command syntax and capabilities are nearly identical in each environment. To display the syntax, enter `eld` at the command prompt.

For more details, see the:

- *DLL Programmer’s Guide for TNS/E and TNS/X Systems*
- *eld and xld Manual*

- *enoft Manual*
- *rld Manual*

Native Inspect Symbolic Debugger

The Native Inspect symbolic debugger is an interactive tool that enables you to identify and correct programming errors in programs.

You can use Native Inspect to:

- Step through code
- Set breakpoints
- Display source
- Display variables

For more details on debugging capabilities, see the *Native Inspect Manual*.

If you have installed the NSDEE Core with Debugging product, you can also use it to identify and correct programming errors in programs. For more information about NSDEE, see the online help for the software.

Visual Inspect Symbolic Debugger

Visual Inspect is an optional PC-based (GUI) symbolic debugger designed to work in complex distributed environments. Visual Inspect:

- Supports application debugging in either the development or production environment
- Uses program visualization, direct manipulation, and other techniques to improve productivity for both new and sophisticated users
- Provides source-level debugging for servers executing in NonStop environment; provides additional application navigation features that allow a higher level of abstraction
- Supports both CISC and native machine architectures and compilers (that is, C, C++, COBOL, TAL, EpTAL, pTAL, D-series Pascal, and FORTRAN), Guardian and OSS executing environments
- Is available also as a stand-alone product

For more details about enabling and using Visual Inspect, see the Visual Inspect online help. See also the descriptions of pragmas **INSPECT** on page 256 and **SYMBOLS** on page 315.

NOTE:

Visual Inspect debugger is not supported on TNS/X systems.

TNS/E Native Object File Tool (enoft Utility)

The native object file tool (the `enoft` utility) reads and displays information about TNS/E native object files. You can use `enoft` to:

- Determine the optimization level of procedures
- Display object code with corresponding source code

- List object file attributes
- List unresolved references

`enoft` runs in the Guardian and OSS environments. The `enoft` syntax and capabilities are nearly identical in each environment. For more details, see the *enoft Manual.*

Native object files are in Executable and Linking Format (ELF), a standard format used for object files, with HPE extensions. For more details on native object files, see the *eld Manual.*

A native object file is either a linkfile or a loadfile, but not both. The native compilers produce only linkfiles, while the `nld` utility can produce either linkfiles or loadfiles.

Linkfiles can be used again as input to `eld`. Loadfiles can be used as input to `eld` only for modifying loadfile attributes. Both linkfiles and loadfiles can be used as `enoft` input.

	Can Be Linked to Produce a Loadfile	**Can Be Executed**
Linkfiles	Yes	No
Loadfiles	No	Yes

NonStop SQL/MP Compiler and NonStop SQL/MX Compiler

The NonStop SQL/MP compiler processes embedded SQL statements from C source programs and generates the correct NonStop SQL/MP database calls. For more details see the *SQL/MP Programming Manual for C*.

The NonStop SQL/MX compiler, available on the OSS platform, processes embedded SQL statements from TNS/E native C or C++ source programs and generates the correct NonStop SQL/MX database calls. For more details, see the *SQL/MX Programming Manual for C and COBOL*.

TNS/E Native C and C++ Migration Tool

The native C and C++ migration tool, NMCMT, scans source files and produces a diagnostic listing. The listing identifies most C and C++ language source code changes required to migrate from TNS C (D20 or later product versions) to native C or C++.

The native mode migration tool is available in both the Guardian and OSS environments, and it is integrated into ETK on the PC.

TNS/E Code Profiling Utilities

The TNS/E Code Profiling Utilities provide these capabilities:

- Evaluate the code coverage provided by application test cases. The utilities use information provided by a specially-instrumented object file to produce a report that indicates which functions and basic blocks were executed, and how many times each was executed.
- Optimize an application through a process called profile-guided optimization. In profile-guided optimization, a specially-instrumented object file is executed to produce a data file containing information about the execution path of the program. That data file, along with the original source code, is then input to the compiler to generate more efficient object code.

Using the Code Profiling Utilities requires a special compilation to produce an object file containing the required instrumentation. To create such an object file, specify the **CODECOV** on page 214 or **PROFGEN** on page 287 option on the compiler command line. Several other compiler options are also related to code profiling. These are the **PROFDIR** on page 284, **PROFUSE** on page 288, and **BASENAME** on page 205 options.

The Code Profiling Utilities can be used with native TNS/E applications.

NOTE:

The Code Profiling Utilities are intended for data generation and collection in a test environment only. The use of instrumented object code is not recommended for production environments. Applications compiled with code profiling instrumentation will experience greatly reduced performance.

For details on using the Code Profiling Utilities, see the *Code Profiling Utilities Manual*.

Features of TNS/E Native C and C++

Beginning with the H06.01 RVU, TNS/E native C and C++ provides:

- The language as defined by *The Annotated C++ Reference Manual* by Ellis and Stroustrup (excluding support for exception handling).
- The language specification in the 1996 *X3J16/WG21 Working Paper*. VERSION2 of the native C++ compiler is based on the 1996 standard, including support for exceptions and features that are not in the *The Annotated C++ Reference Manual* but are new in the working paper. A summary of those major features appears in the description of pragma **VERSION2** on page 324 . A list of specific features accepted from the working paper appears in **Features and Keywords of Version 2 Native C++** on page 565.
- VERSION3 of the native C++ compiler, which is based on the 1998 standard. VERSION3 is the default library used by the native C++ compiler.

This manual is not intended to be a reference manual for ANSI C or for C++. For a complete description of ANSI C, see ANSI X3.159. For a complete description of C++, see ANSI X3J16/96-0225 (VERSION2) and ISO/IEC 14882:1998(E) (VERSION3).

Useful references on C and C++ include:

- *ANSI C*, American National Standards Institute. ANSI X3.159-1989.
- Ellis, Margaret A. and Stroustrup, Bjarne. *The Annotated C++ Reference Manual*. Addison Wesley, 1990.
- ISO/IEC. *Programming Languages – C*. International Standard ISO/IEC 9899. First edition 1990-12-15.
- *Working Paper for Draft Proposed International Standard for Information Systems–Programming Language C++*. X3, Information Processing Systems. 2 Dec 1996. X3J16/96-0225 WG21/N1043 (the standard on which the NonStop VERSION2 of the Standard C++ Library is based).
- International Standard ISO/IEC 14882:1998(E) Programming Languages -- C++ (the 1998 standard on which the NonStop VERSION3 of the Standard C++ Library is based).
- International Standard ISO/IEC 14882:2003(E) Programming Languages -- C++ (a newer standard).

HPE includes several extensions to ISO/ANSI standard C that make C an effective language for writing applications that execute under the HPE NonStop OS. Some of these features are:

- Access to several types of physical files

  There is an extensive set of I/O library routines that enable you to access many different types of physical files, including:

    - C disk files, which are odd-unstructured files and have a file code of 180
    - EDIT disk files, which have a file code of 101
    - Processes
    - Terminals
    - $RECEIVE

- Two file-reference models

  There are two sets of input and output routines; each set has its own method of tracking, maintaining, and referring to a file. These methods are called file-reference models, they are:

    - The ANSI model, which uses FILE pointers to identify files
    - The alternate or UNIX-style model, which uses file descriptors to identify files

      With two file-reference models available, you can select the model whose I/O services best suit the needs of your application. For more details, see **Using the C Run-Time Library** on page 78.

- Access to Guardian system procedures Call procedures in the Guardian system library using the `cextdecs` header file in the Guardian environment or the `cextdecs.h` header file in the OSS environment and PC environment.
- Access to Open System Services system calls and library calls Call routines that are part of the Open System Services library.
- Access to procedures written in other languages TNS programs can call procedures written in C, C++, TNS COBOL, FORTRAN, D-series Pascal, and TAL. Native programs can call procedures written in C, C++, native COBOL, and pTAL.
- Access to a NonStop SQL/MP database or a NonStop SQL/MX database Your TNS/R native C or C++ program can interface to a NonStop SQL/MP or NonStop SQL/MX database using embedded SQL.
- Fault-tolerant programs Write fault-tolerant process pairs using the active backup programming model.

TNS/X Native C and C++ Language System

The TNS/X native C and C++ language system generates TNS/E and TNS/X native C and C++ programs for the Guardian and OSS environments.

Components of this language system:

- **TNS/X Native C Compiler** on page 42
- **TNS/X Native C++ Compiler** on page 42
- **TNS/X Native C Run-Time Library** on page 43
- **C++ Run-Time Library and Standard C++ Library** on page 43
- **TNS/X Native Linker (xld Utility)** on page 44
- Optional components:

    - **Native Inspect Symbolic Debugger**
    - **TNS/X Native Object File Tool (xnoft Utility)** on page 45
    - **NonStop SQL/MP Compiler and NonStop SQL/MX Compiler** on page 46
    - **TNS/X Native C and C++ Migration Tool** on page 46

TNS/X Native C Compiler

The TNS/X native C compiler accepts:

- C language source files that comply with the ISO/ANSI C Standard (ISO/IEC 9899:1990, Programming Languages–C or ANSI C X3.159-1989, Programming Language C) or Common-Usage C (sometimes called Kernighan and Ritchie C, or K&R C). Starting with L16.05 RVU, TNS/X native C compiler accepts C language source files that comply with ISO/IEC 9899:2011.
- The compiler accepts programs that conform to either the 1989 or the 1999 ISO C standard. The default is 1989.

Complex types and several new math functions and macros are not supported for Tandem floating point format, but all other c99 features are available when compiled using the Tandem floating point format option. For more information about this support, see **c99 Full Support** on page 602.

NOTE: To use these features:

- When compiling on Guardian, use the C99 option of the CCOMP command to enable compiling to the 1999 standard (for example `CCOMP/ in filec/;c99`. The default is to compile according to the 1989 standard.
- When compiling on OSS and Windows, use `c89` to compile using the 1989 standard or use `c99` to compile using the 1999 standard.

HPE NonStop extensions that support the native architecture.

The TNS/X native C compiler can be run in the Guardian and OSS environments:

- In the Guardian environment, use the CCOMP command to run the TNS/X native compiler. For syntax information, see **Compiling a C Module** on page 338 .
- In the OSS environment, use the native `c89` utility, or the `c99` utility or the `c11` utility to run the TNS/X native C compiler. For syntax information, see the `c89(1)` or the `c99(1)` reference page either online or in the *Open System Services Shell and Utilities Reference Manual*. The *Open System Services Programmer's Guide* provides guidance on the use of C in the OSS environment.
- On a PC running the Windows operating system, use NSDEE to compile C code. You can also use the command-line cross compiler (named `c89` or `c99` or `c11`) outside NSDEE. For more details, see the NSDEE online help.

TNS/X Native C++ Compiler

TNS/X native C++ compiler accepts one of three dialects of the C++ language. The versions accept C++ language source files and support HPE language extensions. However, the versions support different standards as described in the descriptions of pragmas **VERSION4**, **VERSION3**, and **VERSION2**.

The TNS/X native C++ compiler can be run in the Guardian and OSS environments:

- In the Guardian environment, use the `CPPCOMP` command to run the native C++ compiler. For syntax information, see **Compiling and Linking Native C and C++ programs**.
- In the OSS environment, use the native `c89`, `c99`, or `c11` utility to run the TNS/X native C++ compiler. For syntax information, see the `c89(1)`, `c99(1)`, or `c11(1)` reference page either online or in the *Open System Services Shell and Utilities Reference Manual*. The *Open System Services Programmer's Guide* provides guidance on the use of C in the OSS environment.
- On a PC running the Windows operating system, use NSDEE to compile C and C++ code. You can also use the command-line cross compiler (named `c89` or `c99` or `c11`) outside NSDEE. For more details, see *NSDEE online help*.

The native C++ compiler provides a powerful and simplified development environment. For example, to create an executable native C++ program, you run only the native C++ compiler and the `xld` native linker.

TNS/X Native C Run-Time Library

The native C run-time library provides functions conforming to the ISO/ANSI C Standard. It also contains functions conforming to the X/OPEN UNIX 95 specification and HPE extensions to these standards.

The native C run-time library supports Guardian and OSS processes. The native C run-time library is nearly identical for the Guardian and OSS environments and therefore increases the interoperability between environments. For more details on interoperability, see the *Open System Services Programmer's Guide*.

The native C run-time library provides locale-sensitive functions and algorithmic code-set converters for use in internationalized OSS applications. For more details, see the *Software Internationalization Guide*.

C++ Run-Time Library and Standard C++ Library

The C++ run-time library and the Standard C++ Library are available to every C++ program. However, there are three versions of the libraries, as listed in this subsection. Specifying a version establishes a context that includes the dialect of the native C++ compiler, the run-time libraries available, and the libraries that are automatically linked when compile a executable object file.

VERSION2 Standard C++ Library

For C++ **VERSION2** on page 324, these libraries are available:

- The HPE NonStop C run-time library (file XCRTLDLL)
- The HPE NonStop C++ common run-time library (product T2831, file XCPPCDLL)
- These libraries depend upon the NonStop 32-bit CRE library (file XCREDLL)
- The VERSION2-specific draft Standard C++ Library from Rogue Wave (product T2832, file XCPP2DLL)
- Tools.h++ version 7.0 (product T2835, file XTLH7DLL)

VERSION3 Standard C++ Library ISO/IEC

For C++ **VERSION3** on page 326 (the default version), these libraries are available:

- The HPE NonStop C Runtime Library (file XCRTLDLL for 32-bit programs and WCRTLDLL for 64-bit programs)
- The VERSION3 library, which includes:
  - The HPE NonStop C++ common run-time library (product T2831, files 32-bit XCPPCDLL and 64-bit WCPPCDLL)
  - The VERSION3-specific ratified Standard C++ Library based on the standard document: ISO/IEC 14882:2003 (product T2833, files 32-bit XCPP3DLL and 64-bit WCPP3DLL).
- The 32-bit versions of the libraries depend upon NonStop CRE library, XCREDLL and the 64-bit versions of the libraries depend upon WCREDLL. WCPP3DLL depends upon WCPPCDLL.

VERSION4 Standard C++ Library ISO/IEC

For C++ **VERSION4**, these libraries are available:

NOTE: VERSION4 is the default version when using c11 for compiling the programs. VERSION3 is the default version when using c89, c99, or CPPCOMP for compiling the programs.

- The HPE NonStop C Runtime Library (file XCRTLDLL for 32-bit programs and WCRTLDLL for 64-bit programs)
- The VERSION4 library, which includes:
  - The HPE NonStop C++ common C++ library (product T2831, files 32-bit XCPPCDLL and 64-bit WCPPCDLL)
  - The VERSION4-specific ratified Standard C++ Library ISO/IEC LLVM libc++, based on the standard document: ISO/IEC 14882:2011 (product T2837, files 32-bit XCPP4DLL and 64-bit WCPP4DLL).
- The 32-bit versions of the libraries depend upon NonStop CRE library, XCREDLL and the 64-bit versions of the libraries depend upon WCREDLL. WCPP4DLL depends upon WCPPCDLL.
- For threaded VERSION4 programs, the C++ thread library (XCPPTDLL, WXPPTDLL) is required in addition to the XCPP4DLL or WCPP4DLL library.

C++ Header Files

The C++ header files for the three versions are combined in product T2830. Built-in checking enforces the use of the correct header files with each version.

Tools.h++ version 7 header files are found in product T2834.

TNS/X Native Linker (xld Utility)

The `xld` native linker links one or more TNS/X native object files (files generated by the native compilers or `xld`) to produce either an executable (a loadfile) or relinkable native object file (linkfile). `xld` can also modify process attributes, such as HIGHPIN, of executable native object files and remove nonessential information from native object files.

The `xld` native linker processes PIC (position-independent code), the format necessary when using dynamic-link libraries (DLLs).

`xld` cannot replace individual procedures and data blocks in an object file or build an object file from individual procedures and data blocks. `xld` operates on procedures and data blocks, but only in terms of an entire object file.

`xld` runs in the Guardian and OSS environments in addition to NSDEE on the Windows system. You can specify the target platform as NonStop. Command syntax and capabilities are nearly identical in each environment. To display the syntax, enter `xld` at the command prompt.

For more details, see the:

- *DLL Programmer's Guide for TNS/E and TNS/X Systems*
- *eld and xld Manual*
- *xnoft Manual*
- *rld Manual*

Native Inspect Symbolic Debugger

The Native Inspect symbolic debugger is an interactive tool that enables you to identify and correct programming errors in programs.

You can use Native Inspect to:

- Step through code
- Set breakpoints
- Display source
- Display variables

For more details on debugging capabilities, see the *Native Inspect Manual*.

If you have installed the NSDEE Core with Debugging product, you can also use it to identify and correct programming errors in programs. For more information about NSDEE, see the online help for the software.

TNS/X Native Object File Tool (xnoft Utility)

The native object file tool (the `xnoft` utility) reads and displays information about TNS/X native object files. You can use `xnoft` to:

- Determine the optimization level of procedures
- Display object code with corresponding source code
- List object file attributes
- List unresolved references

`xnoft` runs in the Guardian and OSS environments. The `xnoft` syntax and capabilities are nearly identical in each environment. For more details, see the *xnoft Manual.*

Native object files are in Executable and Linking Format (ELF), a standard format used for object files, with HPE extensions. For more details on native object files, see the *eld and xld Manual.*

A native object file is either a linkfile or a loadfile, but not both. The native compilers produce only linkfiles, while the `nld` utility can produce either linkfiles or loadfiles.

Linkfiles can be used again as input to `xld`. Loadfiles can be used as input to `xld` only for modifying loadfile attributes. Both linkfiles and loadfiles can be used as `xnoft` input.

	Can Be Linked to Produce a Loadfile	**Can Be Executed**
Linkfiles	Yes	No
Loadfiles	No	Yes

NonStop SQL/MP Compiler and NonStop SQL/MX Compiler

The NonStop SQL/MP compiler processes embedded SQL statements from C source programs and generates the correct NonStop SQL/MP database calls. For more details see the *SQL/MP Programming Manual for C*.

The NonStop SQL/MX compiler, available on the OSS platform, processes embedded SQL/MX code from TNS/X native C or C++ source programs and generates the correct NonStop SQL/MX database calls. For more details, see the *SQL/MX Programming Manual for C and COBOL*.

TNS/X Native C and C++ Migration Tool

The native C and C++ migration tool, NMCMT, scans source files and produces a diagnostic listing. The listing identifies most C and C++ language source code changes required to migrate from TNS C (D20 or later product versions) to native C or C++.

The native mode migration tool is available in both the Guardian and OSS environments.

TNS/X Code Profiling Utilities

The TNS/X Code Profiling Utilities provide these capabilities:

- Evaluate the code coverage provided by application test cases. The utilities use information provided by a specially-instrumented object file to produce a report that indicates which functions and basic blocks were executed, and how many times each was executed.
- Optimize an application through a process called profile-guided optimization. In profile-guided optimization, a specially-instrumented object file is executed to produce a data file containing information about the execution path of the program. That data file, along with the original source code, is then input to the compiler to generate more efficient object code.

Using the Code Profiling Utilities requires a special compilation to produce an object file containing the required instrumentation. To create such an object file, specify the **CODECOV** on page 214 or **PROFGEN** on page 287 option on the compiler command line. Several other compiler options are also related to code profiling. These are the **PROFDIR** on page 284, **PROFUSE** on page 288, and **BASENAME** on page 205 options.

The Code Profiling Utilities can be used with native TNS/X applications.

NOTE:

The Code Profiling Utilities are intended for data generation and collection in a test environment only. The use of instrumented object code is not recommended for production environments. Applications compiled with code profiling instrumentation will experience greatly reduced performance.

For details on using the Code Profiling Utilities, see the *Code Profiling Utilities Manual.*

Features of TNS/X Native C and C++

TNS/X native C and C++ provides:

- The language as defined by *The Annotated C++ Reference Manual* by Ellis and Stroustrup (excluding support for exception handling).
- The language specification in the 1996 *X3J16/WG21 Working Paper*. VERSION2 of the native C++ compiler is based on the 1996 standard, including support for exceptions and features that are not in the *The Annotated C++ Reference Manual* but are new in the working paper. A summary of those major features appears in the description of pragma **VERSION2** on page 324. A list of specific features accepted from the working paper appears in **Features and Keywords of Version 2 Native C++** on page 565.
- VERSION3 of the native C++ compiler, which is based on the 1998 standard. VERSION3 is the default library used by the native C++ compiler.
- VERSION4 of the native C++ compiler, which is based on the 2011 standard. VERSION4 is available only on TNS/X systems

This manual is not intended to be a reference manual for ANSI C or for C++. For a complete description of ANSI C, see ANSI X3.159. For a complete description of C++, see ANSI X3J16/96-0225 (VERSION2) and ISO/IEC 14882:1998(E) (VERSION3).

Useful references on C and C++ include:

- *ANSI C*, American National Standards Institute. ANSI X3.159-1989.
- Ellis, Margaret A. and Stroustrup, Bjarne. *The Annotated C++ Reference Manual*. Addison Wesley, 1990.
- ISO/IEC. *Programming Languages – C*. International Standard ISO/IEC 9899. First edition 1990-12-15.
- *Working Paper for Draft Proposed International Standard for Information Systems–Programming Language C++*. X3, Information Processing Systems. 2 Dec 1996. X3J16/96-0225 WG21/N1043 (the standard on which the NonStop VERSION2 of the Standard C++ Library is based).
- International Standard ISO/IEC 14882:1998(E) Programming Languages -- C++ (the 1998 standard on which the NonStop VERSION3 of the Standard C++ Library is based).
- International Standard ISO/IEC 14882:2003(E) Programming Languages -- C++ (a newer standard).

HPE includes several extensions to ISO/ANSI standard C that make C an effective language for writing applications that execute under the HPE NonStop OS. Some of these features are:

- Access to several types of physical files

  There is an extensive set of I/O library routines that enable you to access many different types of physical files, including:

  - C disk files, which are odd-unstructured files and have a file code of 180
  - EDIT disk files, which have a file code of 101
  - Processes

    - Terminals
    - $RECEIVE

- Two file-reference models

  There are two sets of input and output routines; each set has its own method of tracking, maintaining, and referring to a file. These methods are called file-reference models, they are:

    - The ANSI model, which uses FILE pointers to identify files
    - The alternate or UNIX-style model, which uses file descriptors to identify files

      With two file-reference models available, you can select the model whose I/O services best suit the needs of your application. For more details, see **Using the C Run-Time Library** on page 78.

- Access to Guardian system procedures

  Call procedures in the Guardian system library using the `cextdecs` header file in the Guardian environment or the `cextdecs.h` header file in the OSS environment and PC environment.

- Access to Open System Services system calls and library calls

  Call routines that are part of the Open System Services library.

- Access to procedures written in other languages

  TNS programs can call procedures written in C, C++, TNS COBOL, FORTRAN, D-series Pascal, and TAL. Native programs can call procedures written in C, C++, native COBOL, and pTAL.

- Access to a NonStop SQL/MP database or a NonStop SQL/MX database

  Your TNS/R native C or C++ program can interface to a NonStop SQL/MP or NonStop SQL/MX database using embedded SQL.

- Fault-tolerant programs

  Write fault-tolerant process pairs using the active backup programming model.

Writing Portable Programs

A portable application is an application designed using open, industry-standard languages, application program interfaces (APIs), and middleware, such as the C language and POSIX.1 API. A portable application can be moved between hardware systems and middleware environments from different vendors. This subsection provides guidelines for writing portable C programs.

Complying With Standards

Writing programs to international standards enables you to move them between different hardware and software environments with little effort. Write your C programs to comply with the ISO/ANSI C standard: ISO/IEC 9899:1990, Programming Languages–C plus features from 1999 update to this standard (see **c99 Selected Features (C99LITE)** on page 598 for TNS/E programs on H06.21 and later H-series RVUs or for J06.10 and later J-series RVUs, see **c99 Full Support** on page 602 for TNS/E and TNS/X native applications for H06.21 and later H-series RVUs, J06.10 and later J-series RVUs, and L-series RVUs, and see **c11 Support** for TNS/X Compilers conform to the 2011 ANSI C Standard starting from L16.05

RVU. c11 supports compilation of C and C++ programs). For a complete description of ANSI C, see ANSI X3.159.

If you are writing a program to run in the Open System Services (OSS) environment, you should also comply with these standards:

- X/Open Common Applications Environment (CAE) Specification, System Interfaces and Headers, Issue 4, Version 2, 1994.
- X/Open Common Applications Environment (CAE) Specification, System Interface Definitions, Issue 4, Version 2, 1994.
- ISO/IEC 9945-1:1990, Information Technology—Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language].
- ISO/IEC DIS 9945-2:1992, Information Technology—Portable Operating System Interface (POSIX) — Part 2: Shell and Utilities.

The OSS environment provides all the function calls specified in the POSIX.1 standard, and many of the function calls specified in the POSIX.2 standard and the XPG4 specification. Before using a function that is not part of the ISO/ANSI C standard, check to make sure that the OSS environment provides that function.

Feature-test macros enable you to check how well a program complies with these standards. The C header files contain definitions required by the ISO/ANSI C standard, the POSIX.1 standard, the POSIX.2 standard, the XPG4 specification, and the XPG4.2 specification. You control the visibility of these definitions with feature-test macros. If a program does not conform to the standard or specification controlled by a feature-test macro, the compiler issues error and warning messages.

The five feature-test macros that apply to standards compliance are:

`_POSIX_C_SOURCE=1`	Makes visible identifiers required or permitted by the POSIX.1 standard.
`_POSIX_C_SOURCE=2`	Makes visible identifiers required or permitted by the POSIX.1 and POSIX.2 standards.
`_XOPEN_SOURCE`	Makes visible identifiers required or permitted by the XPG4 specification. Represents the OSS compile default.
`_XOPEN_SOURCE_EXTENDED=1`	Makes visible identifiers specified in the XPG4.2 specification as extensions to the XPG4 specification.
`_TANDEM_SOURCE`	Makes visible the identifiers required or permitted by extensions made by HPE. Represents the Guardian compile default for the TNS C compiler. Includes some of the identifiers required by the XPG4 specification, in addition to POSIX.1 and POSIX.2 standards.

The XPG4 specification includes the identifiers required by the POSIX.1 and POSIX.2 standards. Therefore, specifying the `_XOPEN_SOURCE` macro automatically includes the identifiers included by the `_POSIX_C_SOURCE=2` macro.

For more details about feature-test macros, see **Feature-Test Macros** on page 187.

Following Good Programming Practices

In general, the more you follow good programming practices, the easier it will be to port your program to other hardware and software environments. A few of the most important good programming practices are:

- Use strictly conforming C language features as described in the ISO/ANSI C standard as much as possible. Isolate HPE defined and extended C language features into specific modules.
- Use function prototypes.
- Place all environment-specific function declarations in a common header file, such as `environh`.
- Make sure the type of a function’s actual and formal parameters are the same.
- Define every function with an explicit return type. Make sure the type of a return expression is the same as the return type of the function.
- Write your programs to a template similar to:
    - Feature-test macros
    - System headers
    - Local headers
    - Macros
    - File scope variables
    - External variables
    - External functions
    - Structures and unions
    - Signal-catching functions
    - Functions
    - Main function
- Do not write code that relies on processor architecture. Be careful when writing code that relies on word size, pointer size, bit fields, arithmetic precision, byte order, stack size, stack growth, heap size, and heap growth.
- Do not make assumptions about the size and format of any data type:
    - Use type `short` and type `long` instead of type `int`, if possible. In particular, do not interchange between type `int` and type `long`.
    - Do not assign an `int` to or from a pointer without an explicit type cast.
    - Do not assume that different pointer types are the same.
- Use `unsigned` types for bit fields.

There are many commercially available texts that describe how to write portable applications in C, including:

- Horton, Mark. Portable C Software. Prentice Hall, Inc., 1990.
- Jaeschke, Rex. Portability and the C Language. Hayden Books, 1989.
- Lapin, J. E. Portable C and UNIX System Programming. Rabbit Software, 1987.
- Rabinowitz, Henry, and Chaim Schaap. Portable C. Prentice Hall, Inc., 1989.

Porting Programs to HPE C and C++ for NonStop Systems

HPE C complies with the ISO/ANSI C standard. Any C program that strictly conforms with this standard and does not use features beyond this standard can immediately be compiled in HPE C. Programs written in Common-Usage C (also called Kernighan and Ritchie C or K&R C) can also be compiled with the native C compiler.

For more details on porting UNIX C and C++ programs to Open System Services, see the *Open System Services Porting Guide*.

It is impossible to provide a complete set of guidelines on porting programs to ISO/ANSI C, but most porting issues have one of two solutions:

- Replace nonstandard function calls with one or more functions from the ISO/ANSI C standard. In the OSS environment, you can also use many function calls defined in the POSIX.1 and POSIX.2 standards and the XPG4 specification.
- Redesign the code to use functions and features of ISO/ANSI C.

Many UNIX compilers now comply with the ISO/ANSI C standard. Unlike the HPE C compilers, most of these compilers do not strictly enforce the standard by default. These compilers allow features that do not comply with the standard but that the compilers can still process correctly. Therefore, many programs that you thought to be compliant are not.

To compile standard-compliant programs with the native C compiler, specify the pragma `KR` (for Kernighan & Ritchie or Common-Usage C). To compile such programs with the TNS C compiler, first convert the programs to ISO/ANSI C in their original environment. The documentation for these compilers often describes the specific changes required to make a program comply with ISO/ANSI C.

These commercially available texts describe writing programs that comply with the ISO/ANSI C standard and porting from Common-Usage C to ISO/ANSI C:

- Harbison, Samuel P. and Guy L. Steele. *C, A Reference Manual*. Prentice Hall, Inc., 1991.
- Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Languag*e. Prentice Hall, Inc., 1988.
- Strake, David. *C Style: Standards and Guidelines*. Prentice Hall, Inc., 1992.
- *ANSI C Transition Guide*. Prentice Hall, Inc., 1990.

Porting Without Data Alignment Problems

Mixed-Language Programming for TNS/R, TNS/E, and TNS/X Native Programs on page 141, and **Handling TNS Data Alignment** on page 497, provide guidance on the use of compiler pragmas to avoid data alignment problems when sharing data between programs in different compiler languages or when porting programs and data between HPE development environments or between NonStop platforms.

You can also use the Data Definition Language (DDL) to define data structures in a manner that minimizes data alignment problems. DDL can be used to describe Enscribe data file structures or to generate source code with data-object filler appropriate for various compiler data alignment pragmas. For more details, see the *Data Definition Language (DDL) Reference Manual*.

Guardian and OSS Environment Interoperability

Open System Services (OSS) is an alternative to the Guardian interface through which users and programs can interact with the NonStop OS. The main purpose of Open System Services is to provide an open interface to the operating system for supporting portable applications.

An application program can be compiled to run in either the OSS or Guardian environment and can use services from each environment, including the API, tools, and utilities. The interoperability enables you to:

- Compile and link OSS programs in the Guardian environment
- Compile and link Guardian programs in the OSS environment
- Call most Guardian C functions and Guardian system procedures from OSS programs
- Call most OSS functions from Guardian programs

You do not need to do anything special for programs that manipulate objects exclusively in the environment in which they run. However, the NonStop OS enables you to write programs that manipulate objects in both the Guardian and OSS environments.

For TNS C programs, Guardian and G-series OSS C run-time interoperability is supported for programs using only the 32-bit or wide data model. Interoperability is not supported for TNS C programs using the 16-bit data model. Interoperability is supported for all native programs, because native C programs support only the 32-bit or wide data model.

Some of the interoperable modules created before the D44 product version require code changes and recompilation. For more details and instructions, see **Changes Required to Interoperable Compilation Modules at D44** on page 85.

For more details on Guardian and OSS interoperability, see the *Open System Services Programmer's Guide*.

C and C++ Extensions

This chapter describes the language extensions to the ISO/ANSI C standard that are supported by the HPE native C and C++ compilers, the TNS C compiler, and the TNS C++ preprocessor.

Keywords

A keyword is a word symbol that has a special predefined syntactic or semantic meaning in a programming language. The table **Extensions to Reserved Keywords** lists the reserved keywords that HPE has added as extensions to the C and C++ languages. The table **Extensions to Reserved Keywords** notes whether there are differences between using the keyword in the native or TNS environment.

Table 4: Extensions to Reserved Keywords

Keywords With Same Function in Both Native and TNS	Keywords Allowed in Both Native and TNS But With No Effect in Native	Keywords Applicable Only in TNS
`_alias`	`_baddr`	`_cc_status`
`_arg_present`	`_cobol`	
`_bitlength`	`_cspace`	
`_c`	`_far`	
`_extensible`	`_fortran`	
`_optional`	`_lowmem`	
`_resident`	`_near`	
`_tal`	`_pascal`	
`_variable`	`_procaddr`	
	`_waddr`	

Native C and C++ treat these HPE extensions as keywords only if the **EXTENSIONS** on page 227 pragma is in effect. This feature allows you to choose to compile a program so that standards conformance is enforced. This feature also ensures the acceptance of a standard-conforming program that might happen to use one of these keywords as an identifier. Enforcing standards conformance is the default setting. To disable it, specify the EXTENSIONS pragma.

TNS C and C++ always treat these HPE extensions as reserved keywords. There is no way to disable this behavior. This feature might cause some strictly conforming program to fail to compile.

Declarations

This subsection lists all the extensions HPE has made to the *declaration-specifier* syntax defined in the ISO/ANSI C standard. These syntax extensions also apply to C++, because the rules for C are a subset of the rules for C++. In some cases, C++ imposes additional usage considerations.

The *storage-class-specifier*, *type-specifier*, and *type-qualifier* are defined in the ISO/ANSI C standard. Any extensions to these specifiers are described in this subsection. The *attribute-specifier* is a P extension and is described later in this subsection.

```
declaration-specifiers:

   storage-class-specifier  [ declaration-specifiers ]
   type-specifier           [ declaration-specifiers ]
   type-qualifier           [ declaration-specifiers ]
   attribute-specifier      [ declaration-specifiers ]
```

Storage Class Specifier

The `_lowmem` and *export-attribute* storage-class specifiers are HPE extensions.

```
storage-class-specifier is one of:

   typedef
   extern
   static
   auto
   register
   export-attribute
   _lowmem
```

`typedef`, `extern`, `static`, `auto`, and `register` are described in the ISO/ANSI C standard.

`_lowmem` specifies that arrays and structures in the declaration are to be stored in the user data space instead of the extended segment (only for TNS C and C++). Consequently, `_lowmem` is useful only when compiling for the large-memory model or the wide-data model. Note that you can use `_lowmem` alone, or you can specify it after the auto, extern, or static storage-class specifiers.

The `_lowmem` storage class specifier is applicable only in the TNS environment. In the native environment, `_lowmem` has no effect on code generation or the storage of objects. It is supported to allow source-level compatibility with TNS C and C++. Source-level compatibility involves only accepting syntactically correct TNS programs, not diagnosing semantic violations with `_lowmem` usage.

Usage Guidelines for _lowmem

- `_lowmem` cannot be used with the `register` storage-class specifier or with the `_cspace` type qualifier.
- `_lowmem` cannot be used with a formal parameter declaration or a function declaration.
- `_lowmem` cannot be used with a class or struct member unless the member is `static`.
- If you specify only `_lowmem`, the default storage classes apply:
    - Variables declared within function bodies and variables declared within blocks have the default storage class `auto`.
    - Variables defined outside of a function have the default storage class `extern`.
    - Functions have the default storage class `extern`.

Examples

This example shows how `_lowmem` affects allocation of data storage using the large-memory model:

```
#pragma XMEM

static int x[5000];
static _lowmem int m[1000];
_lowmem struct triplet
{
  int x,y,z;
} t1;
```

The pragma `XMEM` specifies compilation for the large-memory model. Therefore, storage for the array `x` is allocated in the extended segment. However, because the declarations for the array `m` and the structure `t1` specify `_lowmem`, storage for them is not allocated in the extended segment. Instead, storage is allocated in the user data space.

Export Attribute (Native C and C++ Only)

The `export$` and `import$` keywords are supported only on native C/C++ on TNS/E and TNS/X systems. These keywords export and import functions and data to and from a DLL. They specify whether a defined or declared class is to be exported or imported, respectively.

```
export-attribute:
export$
import$
```

`export$`

indicates that a specification is a definition and its associated members will be exported.

`import$`

indicates that a specification is only a declaration and that the definition will be found through external linkage to a library where the associated members are exported. If applied to a definition, the definition will be treated as a declaration.

Usage Guidelines for the Export Attribute

- The keyword `export$` may only be used with a definition. If specified for a declaration, and its definition is not within the compilation unit, an error is generated. If `export$` is applied to a tentative definition, the compiler treats the tentative definition as a real definition. In C, a tentative definition is a declaration of an identifier of an object that has file scope without an initializer, and without the storage class specifier `extern`.
- The keyword `import$` may only be used with a declaration. If specified for a definition, an error is generated. If it is applied to a tentative definition, the compiler treats the tentative definition as a declaration.
- If a single module in a program has both an `import$` and an `export$` attribute specified for the same function or object, the `export$`attribute takes precedence over the `import$` attribute. The compiler generates a warning.
- These keywords cannot be applied to static functions, static data, or auto data. The compiler issues an error for these assignments.

Examples for Native C/C++

For a complete programming example using `export$` and `import$` and producing a DLL, see **Examples** on page 392.

```
import$ extern int I;      /* OK--this is a data declaration*/
export$ extern int j = 1;  /*OK--this is a data definition*/
export$ static int si;     /*Error:  static data */
import$ static int si = 0; /* Error: static data */
import$ extern int j;      /* Warning: already specified as export$*/
import extern int i1 = 1;  /* Error:  this is a definition*/
export$ extern int i2;     /* Error if i2 is not defined later*/
export$ int i3;            /*Causes tentative def to be treated as a real def*/
import$ int i4;     /*Causes tentative def to be treated as declaration*/
export$ static foo1 () {}  /*Error: static function */
import$ static foo2 () {}  /*Error:static function */
export$ extern foo3 () {}  /* OK - extern function definition */
export$ extern foo4 ();    /*Error: if foo4 not defined later*/
import$ extern foo5 ();    /*OK - extern function declaration*/
import$ extern foo6 () {}  /*Error: extern function definition */
```

Usage Guidelines for the Export Attribute (Native C++ Only)

- In C++, `export$` and `import$` can be specified for class definitions, in addition to for member functions and static data members. When given for a class definition, these keywords specify that the entire class is either exported or imported, including any Computer Generated class-specific data (CGD). When given for specific members, only those members are exported or imported. Within a single compilation unit, a class is not allowed to both export and import members.
- When you declare a class `export$`, all of its member functions, static data members, and any CGDs are exported. Definitions for these member functions and static data members must be provided. An error is issued for missing definitions. Exported member functions are never inlined.
- Selectively mark member functions and static data members as `export$` or `import$` only if the entire class is marked as neither `export$` nor `import$`. The compiler generates an error if this rule is violated.
- Selective member export/import is best used for providing a class interface that is more restrictive, in which fewer members are exposed to the class's clients.
- If any member of a class is exported or imported, all of its CGDs are also exported or imported, respectively.
- Only members that are introduced by the class are exported or imported; methods inherited by the class are not exported or imported.
- Do not export a class definition in more than one compilation unit or load unit, and do not mix explicit `export$`/`import$` usage with default class definitions. That is, linking or loading errors, or erroneous run-time type checking can occur if either one of these two paradigms is not followed:
    - Exactly one compilation unit (and one load unit) explicitly exports the class (wholly or selectively), and all others explicitly import it.
    - Every compilation unit defining the class uses a default definition (neither `export$` nor `import$`).

Examples of the Export Attribute (Native C++ Only)

```
// export everything from a class

class export$ C1 {
public:
  int I;                        // class data, not exported
  int foo (void) {returnI ;}  // Exported member function
  static int J;               // Exported static data member
};

int C1::J = 1; // Definition for exported static data member

// import everything from a class

class import$ C2 {
public :
  int I;                       // Class data, not imported
  int foo (void) {return I;} // Imported member function,
                             // definition ignored
  static int J;              //Imported static data member
};

int C2::J = 1;  // Error:  definition for imported static data member


//selective exporting/importing


class C3a {

public :
  int I;
  int foo (void) {return I;}  // Member function -
                              // neither exported nor imported
  export$ static int J;       // Exported static data member
};

int C3a::J = 1;  //Definition for exported static data member

class C3b {

public :

  int I;
  int foo (void) {return I;}  // Member function -
                              // neither exported nor imported
import$ static int J;         // Imported static data member
};

// Error to selectively export/import members of an exported/imported class


class export$ C4 {

public :

  int I;
  import$ int foo (void) {return I;}  // Error
  export$ static int J;               // Error
};

int C4::J = 1;  //Definition for exported static data member.

//Error to export & import members from a class in a single compilation unit


class C5 {

public :

  int I;
  export$ int foo (void) {return I;}   // Exported member function
  import$ static int J;                // Error

};
```

Type Specifier

For the TNS compilers, the `_cc_status` type specifier is a HPE extension. For the native compilers, `_cc_status` is a typedef in the `tal.h` header file, and you must include this header file to use `_cc_status`.

NOTE: Do not write C functions or TAL procedures that set a condition code and use `_cc_status` because that is an outdated programming practice. Guardian system procedures retain this interface for reasons of compatibility only.

```
type-specifier
   void | char | short | int | long | float | double|
   signed | unsigned |  bool | _cc_status |
   struct-or-union-specifier | enum-specifier | typedef-name
```

`void`, `char`, `short`, `int`, `long`, `float`, `double`, `signed`, `unsigned`, and `bool`

are as described in the ISO/ANSI C standard. Additional information concerning these predefined data types is given later in this section and in **HPE C Implementation-Defined Behavior** on page 515. Note that the `bool` type became available at the D45 release, but it is available only with native C++ using the **VERSION2** on page 324 directive.

`_cc_status`

indicates that a procedure does not return a value but does set a condition code.

`struct-or-union-specifier`*, *`enum-specifier`* , and *`typedef-name`

are described in the ISO/ANSI C standard.

Usage Guidelines

- If `_cc_status` is used as the return type for a function declaration, the language specifier, if used, must be one of `_c`, `_tal`, or `_unspecified`.
- The `tal.h` header contains six macros that interrogate the results of a function declared with the `_cc_status` type specifier. These macros are:

- For native C and C++:

  ```
  #define _status_lt(x) ((x) < 0)
  #define _status_le(x) ((x) <= 0)
  #define _status_eq(x) ((x) == 0)
  #define _status_ge(x) ((x) >= 0)
  #define _status_gt(x) ((x) >  0)
  #define _status_ne(x) ((x) != 0)
  ```

- For TNS C and C++:

  ```
  #define CCL     2
  #define CCE     1
  #define CCG     0
  #define _status_lt(x) ((x) == CCL)
  #define _status_le(x) ((x) != CCG)
  #define _status_eq(x) ((x) == CCE)
  #define _status_ge(x) ((x) != CCL)
  #define _status_gt(x) ((x) == CCG)
  #define _status_ne(x) ((x) != CCE)
  ```

- The `tal.h` header for TNS C and C++ still supports the `CCL`, `CCE`, and `CCG` macros. These macros have these meaning:

`CCL`	Condition-code register less than zero
`CCE`	Condition-code register equal to zero
`CCG`	Condition-code register greater than zero

- For more details on using `_cc_status`, see **Mixed-Language Programming for TNS Programs** on page 113 , or **Mixed-Language Programming for TNS/R, TNS/E, and TNS/X Native Programs** on page 141.

Examples

This example shows the difference between using the new macros defined at the D40 release and those used prior to the D40 release to examine `_cc_status`.

```
_tal _extensible _cc_status READX ( ... );
#include <tal.h>
...
_cc_status cc;

cc = READX ( ... );

/* Pre-D40 method */
if (cc == CCL) {
...
} else if (cc == CCG) {
...
}

/* D40 method */
if (_status_lt(cc)) {
...
} else if (_status_gt(cc)) {
```

```
...
}
```

Type Qualifier

The _cspace type qualifier is an HPE extension. The _cspace type qualifier is applicable only in the TNS Guardian environment. _cspace is unnecessary in the native environment; however the native compilers recognize the _cspace keyword to provide source-level compatibility with the TNS compiler. Source-level compatibility involves only accepting syntactically correct TNS programs, not diagnosing semantic violations with _cspace usage.

```
type-qualifier

   const  | _cspace | volatile
```

const

is described in the ISO/ANSI C standard.

_cspace

indicates that the specified constant is stored in the current code space (only for TNS C and C++). The _cspace type qualifier must be accompanied by the const type qualifier. For more details on the use of the _cspace type qualifier, see **System-Level Programming** on page 166.

volatile

In the ISO/ANSI C standard, if a variable is declared with the volatile type qualifier, the variable can be modified by an agent external to the program. The architecture of HPE NonStop systems does not support external modifications to variables by agents external to the program. For compatibility with the ISO/ANSI C standard, HPE defines volatile to mean:

- For the TNS C compiler and the TNS C++ preprocessor, the volatile type qualifier is accepted and ignored.
- For the native C and C++ compilers, if a variable is qualified with the volatile type qualifier, this means that loads and stores of this variable must access memory, not a register.

For TNS C++, you cannot use volatile to distinguish specific instances of overloaded member functions.

GCC compatible language extensions

NOTE: These extensions are supported only by the TNS/X compiler.

Zero-length arrays

Zero-length arrays is a language extension which is used to implement variable-length arrays in structures. This extension is enabled only when compiling with EXTENSIONS (Guardian) and -Wextensions (OSS and Windows).

```
#include <stdlib.h>
struct line
 {
size_t length;
char contents[0];
};
struct line * new_line (size_t num_chars)
```

```
{
struct line *Line = (struct line *)
malloc (sizeof (struct line) + num_chars);
Line->length = num_chars;
return Line;
}
```

Statement expressions

A statement expression is a compound statement that is enclosed in parenthesis. The compound statement must have an expression at the end followed by a semicolon. The value of this sub-expression serves as the value of entire construct. (If any other kind of statement is used at the end within parentheses, the construct has type `void`, and thus effectively no value).

This extension is enabled only when compiling with `EXTENSIONS` (Guardian) and `-Wextensions` (OSS and Windows).

```
#define max(a,b) \ ({__typeof(a) _a = (a); __typeof(b) _b = (b);c_a > _b ? _a : _b; })
int foo(void);
int bar(void);
void func(void)
{
int i = max(foo(), bar());
}
```

Pointer arithmetic

This extension supports Addition and subtraction operations on pointers to void and pointers to functions. These operations are performed by considering the size of a void or a function as 1. This extension is enabled only when compiling with `EXTENSIONS` (Guardian) and `-Wextensions` (OSS and Windows).

Example:

```
void * add (void *x, int y) {
return x+y;
}
```

Attribute Specifier

The attribute specifier is an HPE extension. It is used for writing system-level C and C++ functions and for declaring external routines written in other languages.

NOTE: The attribute specifier is an outdated syntax for mixed-language programming and is maintained only for compatibility. To declare external routines written in another language, you should use the `FUNCTION` pragma syntax as described in **FUNCTION** on page 239.

The attribute specifier is applicable only in the Guardian environment.

```
attribute-specifier:
   [ language-specifier ] [ attribute [ attribute ]... ]

language-specifier is one of:
   _c | _cobol | _fortran | _pascal | _tal | _unspecified


attribute is one of:
   _alias ( "external-name " ) |
   _extensible [ ( param-count ) ]
   _resident | _variable
```

language-specifier

A *language-specifier* for a function prototype specifies the language of the body of the external routine being declared.

Considerations for the Native Compilers Only

Native C and C++ support mixed-language programs with modules written in the C, C++, native COBOL85, and native pTAL languages. Therefore, the only language specifiers that apply are`_c`, `_cobol`, _tal, and `_unspecified`. Use `_tal` to denote the pTAL language on TNS/X, TNS/E, and TNS/R systems.

Considerations for the TNS Compilers Only

- TNS C and C++ support mixed-language programs with modules written in C, C++, TNS COBOL, FORTRAN, D-series Pascal, and TAL. Therefore, these language specifiers apply: `_c`, `_cobol`, `_fortran, _pascal, _tal`, and `_unspecified.`
- If you declare an external procedure as `_unspecified`, the actual procedure cannot be both written in C and compiled using the `OLDCALLS` pragma.
- If you declare an external procedure as `_unspecified` and that procedure is actually written in C, in the definition, the name must not contain lowercase alphabetic characters.

Considerations for Both the Native and TNS Compilers

- Only one language specifier can be specified for a function.
- The language specifier is allowed only in function declarations, function definitions, and function pointer declarations. However, `_cobol`, `_fortran`, `_pascal`, `_tal`, and `_unspecified` are not allowed in function definitions. `_c` is allowed in a function definition only if the language being compiled is C.
- For C++, a language specifier is not allowed on overloaded functions, member functions, function templates, inline functions, or functions with default arguments.
- For a function with C-style variable argument lists, the only explicit language attribute allowed is `_c`.
- The language specifier `_unspecified` indicates that the language is unknown and unspecified.

attribute

specifies a function attribute.

Considerations for Both the Native and TNS Compilers

- An *attribute* is allowed only in function declarations, function definitions, and function pointer declarations.
- The same attribute is not allowed to appear more than once in a function declaration.
- In C++, attributes on virtual functions are not inherited.
- A function is allowed to have multiple declarations. If the multiple declarations are incompatible, an error message is generated. If the declarations are compatible, the attribute values are “unioned.” Two declarations for a function are deemed to agree, with respect to the *attributes* , if:

- Either both or neither are specified to be `_extensible` .
- Either both or neither are specified to be `_variable` .
- If both have a *language-specifier* , they must either specify the same language or one *language-specifier* must be `_unspecified` .
- They do not result in any illegal combination as is discussed under the following attribute syntax descriptions.

Effects of Function Attributes table shows the effect of function attributes for each language. “Valid” indicates that the compiler accepts the attribute. “Ignore” indicates that the compiler accepts but ignores the attribute. “Error” indicates that the compiler issues an error.

Table 5: Effects of Function Attributes

Language	No attribute	_alias	_extensible	_resident	_variable
_c	valid	valid	valid	valid	valid
_cobol	valid	valid	error	ignore	error
_fortran	valid	valid	error	ignore	error
_pascal	valid	valid	valid	ignore	error
_tal	valid	valid	valid	ignore	valid

An attribute can be one of these:

`_alias` **("** ***external-name*** **")**

identifies the name of the external routine being declared and is used for mixed-language programming. _alias is used at bind-time for TNS C and C++ programs or at link-time for native C and C++ programs. The *external-name* argument must be enclosed in parentheses and quotation marks, as indicated in the syntax. For example:

```
_alias ("MY^PROC")
```

Use `_alias` to describe the name of an external routine written in COBOL, FORTRAN, D-series Pascal, or TAL that does not have a valid C name. The external-name argument is neither modified (for example, upshifted) in any manner, nor checked to verify that it is a valid identifier for the language of the external routine.

Considerations for both the native and TNS compilers:

- A function pointer is not allowed to have an `_alias` attribute.
- The *external-name* argument cannot be introduced into the program’s namespace.
- For C++, member functions, function templates and overloaded functions are not allowed to have an `_alias` attribute.

`_extensible` **[(** ***param-count*** **)]**

directs the compiler to treat all parameters of the function as though they were optional, even if some parameters are required by your code. You can add new formal parameters to an _extensible function without recompiling callers unless the callers use the new parameter.

Considerations for the native compilers only:

- An `_extensible` function is limited to 31 parameters. If more are supplied an error message is generated.
- `_extensible` and `_variable` are treated the same, `_variable` is a synonym for `_extensible`

Considerations for the TNS compilers only:

A function with no prototype declaration or definition under pragma `OLDCALLS` cannot be `_extensible` .

Considerations for both the native and TNS compilers:

- `_extensible` and `_variable` attributes cannot be specified for the same function.
- C-style variable argument lists (...) are not allowed.
- An `_extensible` attribute cannot be specified for a function that passes a structure by value.
- Returning structures by value is not allowed.
- For `_extensible(` *n* `)` , an error is generated if *n* is greater than the number of formal parameters.
- For C++, these attributes can cause the functions to have extern "C" linkage. That is, all C++ programs that use _extensible or _variable must be explicitly declared with extern "C" linkage. Thus, The function name will not be mangled
  - `_extensible` functions can neither be overloaded nor have default parameters.
  - `_extensible` cannot be specified on global function templates, that is, function templates that are not members of a class.
  - `_extensible` cannot be specified on template member functions, that is, the member functions of a template class.
  - `_extensible` cannot be used on class member functions of any class.

`_extensible` functions in C and C++ are equivalent to extensible procedures in TAL, pTAL, and D-series Pascal. For further discussion of `_extensible` functions in C and C++, see **System-Level Programming** . For further discussion of declaring external routines that are written in pTAL, TAL, or D-series Pascal and that are extensible, see **Mixed-Language Programming for TNS Programs**, or **Mixed-Language Programming for TNS/R, TNS/E, and TNS/X Native Programs**.

`_resident`

causes the function code to remain in main memory for the duration of program execution. The operating system does not swap pages of this code.

Considerations for both the native and TNS compilers:

- The `_resident` attribute is effective only for function definitions. It is ignored if it is given for a function declaration and its corresponding function definition is not in the same compilation unit.
- For C++, if you define a local class within a `_resident` function, all member functions of this local class are implicitly declared to be `_resident` .

`_variable`

directs the compiler to treat all formal parameters of the function as though they were optional, even if some parameters are required by your code.

Considerations for the native compilers only:

`_extensible` and `_variable` are treated the same; `_variable` is a synonym for `_extensible`.

Considerations for the TNS compilers only:

A function with no prototype declaration or definition under pragma `OLDCALLS` cannot be `_extensible` .

Considerations for both the native and TNS compilers:

- `_extensible` and `_variable` attributes cannot be specified for the same function.
- C-style variable argument lists (...) are not allowed.
- A `_variable` attribute cannot be specified for a function that passes a structure by value.
- Returning structures by value is not allowed.
- In C++ these guidelines apply:
  - `_variable` functions can neither be overloaded nor have default parameters.
  - `_variable` cannot be specified on global function templates, that is, function templates that are not members of a class.
  - `_variable` cannot be specified on template member functions, that is, the member functions of a template class.
  - `_variable` cannot be used on class member functions of any class.

Usage Guidelines for Attribute-Specifier Syntax

- The `FUNCTION` pragma is the preferred method for declaring external routines. For more details, see the pragma **FUNCTION** on page 239

  and **Writing Interface Declarations** on page 114 .
- Programs that use this *attribute-specifier* syntax can be made portable by using macro definitions in which the *attribute-specifier* keywords are replaced by nothing.

Examples

This function declaration declares a C or C++ function `myproc`, which is a TAL variable procedure with the externally visible procedure name `My^Proc`:

```
 _tal _variable _alias ("My^Proc") void myproc (short);
```

This example declares a function pointer pointing to a TAL integer extensible procedure:

```
_tal _extensible short (*tal_func_ptr)(void);
```

Pointer Modifiers

The `_baddr`, `_far`, `_near`, `_procaddr`, and `_waddr` pointer modifiers are HPE extensions. They have been added to TNS C and C++ because the TNS architecture has many pointer types, and it is necessary that you have a way to designate what type of pointer is desired. For native C and C++, these pointer modifiers have no effect. They are added to provide source-level compatibility with TNS C and C++. Source-level compatibility involves only accepting syntactically correct TNS programs, not diagnosing semantic violations with pointer modifier usage.

NOTE: The native compilers also recognize the `_ptr32` and `_ptr64` modifiers only on TNS/E and TNS/X systems, for more information see **LP64 Data Model** on page 506.

Pointer for C Programming Language

```
pointer:
   [ modifier-list ] *  [ type-qualifier-list ]
   [ modifier-list ] *  [ type-qualifier-list ] pointer

modifier-list:
   modifier [ modifier-list ]

modifier:
    _far | _near | _baddr | _waddr | _procaddr
```

type-qualifier-list is described in the ISO/ANSI C standard.

Ptr-operator for C++ Programming Language

```
ptr-operator:
   [ modifier-list ] * [ cv-qualifier-seq ]
   [ modifier-list ] & [ cv-qualifier-seq ]
   [ :: ] nested-name-specifier [ modifier-list ] *
          [ cv-qualifier-seq ]

modifier-list:
   modifier [ modifier-list ]

modifier:
   _far | _near | _baddr | _waddr | _procaddr
```

cv-qualifier-seq and *nested-name-specifier* are described in the ANSI C++ standard.

`_far`

denotes a 32-bit extended byte address.

_near

denotes a 16-bit address and points to the lower 32K of the user data segment.

_baddr

modifies `_near`, and denotes that the address is a byte address.

_waddr

modifies `_near`, and denotes that the address is a TNS word address.

_procaddr

denotes a DPCL entry point id in TNS environment.

Usage Guidelines

- For TNS C and C++, the pointer modifiers are independent of the memory model used. A `_near` pointer is 16 bits for both the small and large-memory models. A `_far` pointer is 32 bits for both the small and large-memory models. For native C and C++, the pointer modifiers have no effect and are added only to provide source-level compatibility with TNS C and C++. For native C and C++, all pointers are 32-bit byte addresses.
- The same *modifier* is not allowed to appear more than once in the *modifier-list* for a pointer declaration.
- The *modifiers* `_far` and `_near` are mutually exclusive.
- The *modifiers* _baddr, `_waddr` , and `_procaddr` are mutually exclusive; at most one is allowed.
- If `_near` is used to modify a pointer and neither `_waddr` or `_baddr` is specified, the pointer can contain either a byte address or a TNS word address:

If Type Pointed To Is	Then Pointer Contains
8-bit scalar[1]	byte address
greater than an 8-bit scalar[1]	TNS word address
void	byte address
struct	TNS word address
array	Pointer is the same as a pointer to an element of the array.

[1] Arithmetic types and pointer types are called scalar types. Integral and floating types are collectively called arithmetic types.

- The type qualifier `extptr` is considered obsolete, but TNS C and C++ still support it for `_tal` prototypes. Programs that use this feature will not compile with the native C and C++ compilers.
- For TNS only, certain implicit casting of pointers might result in diagnostics. The following table lists which implicit conversions cause a warning or an error. The column on the far left lists the “from” pointer type and the column headings list the “to” pointer types.

From Type	To Type_far	_near _waddr	_near _baddr	_cspace	_procaddr
far	valid	error	error	warning[1]	error
_near _waddr	valid	valid	warning	warning[1]	error
_near _baddr	valid	warning	valid	warning[1]	error
_cspace	warning[1]	warning[1]	warning[1]	valid	error
_procaddr	error	error	error	error	valid

[1] TNS C++ gives an error for these cases.

- For TNS only, the compiler generates code to perform implicit conversions for all cases in the table, marked valid or warning. However, in the warning cases, the resulting value may lose some information. Explicit casts can be used to override the implicit casts.
- For TNS only, a pointer with a `_baddr` modifier cannot be used to access any TNS word-addressed data types. A pointer with a `_waddr` modifier cannot be used to access any byte-addressed data types.
- For TNS only, pointer arithmetic can be performed only on pointers that point to the same address space. The address spaces are user data space and code space.
- For TNS only, `_procaddr` can be applied only to void *.A pointer declared with the `_procaddr` type qualifier cannot be fused to call a function. The value must be assigned to a typed function pointer and called through that pointer.

Operators

An expression comprises a sequence of operators and operands. An operator specifies an operation that results in a value. The items on which the operator acts are called operands. Operators that act on one operand are referred to as unary operators. Operators that act on two operands are referred to as binary operators. This subsection describes the extensions HPE has made to the operators defined in the ISO/ ANSI C standard.

_arg_present()

The `_arg_present()` operator is an HPE extension. `_arg_present()` is used to determine the presence of a parameter in an extensible or variable function. The compiler generates code to determine whether the supplied operand is present in the actual function call. If the operand is present, 1 is returned; otherwise, 0 is returned.

```
unary-expression:
   _arg_present (formal-parameter-name)
```

formal-parameter-name

defines the operand on which the `_arg_present()` operator operates.

Usage Guidelines

- The `_arg_present()` operator is allowed only in an extensible or variable function.
- The f*ormal-parameter-name* operand in the `_arg_present()` operator must be the name of a formal parameter of the extensible or variable function.

_bitlength()

The `_bitlength()` operator yields the size, in bits, of its operand. Its operand can be an expression or the parenthesized name of a type. The size is determined from the type of the operand, which itself is not evaluated. The result is an integer constant. The `_bitlength()` operator is similar to the sizeof() operator, except that for `_bitlength()` the result is in bits and the *unary-expression* can be a bit-field selection.

```
_bitlength (type-name) | unary-expression
```

type-name

is the name of a type.

unary-expression

is a unary expression.

Usage Guideline

The `_bitlength()` operator cannot be applied to an operand of function type or of incomplete type. If this application is attempted, the result is an unsigned constant whose type is `size_t`.

_optional()

The `_optional()` operator is similar to the pTAL `$OPTIONAL` standard function. It allows you to dynamically pass and omit actual parameters to extensible and variable functions.

The first expression operand is a scalar expression that is evaluated to determine whether an actual argument is to be evaluated and passed to the extensible or variable function. If the first expression is zero, the second expression operand is not evaluated and no actual argument is passed. If the first expression is not zero, the second expression operand is evaluated and passed as the actual argument.

When the `_optional()` operator is used, the extensible and variable mask and count words are computed at run-time. The code generated is slightly slower than using “normal” calls where the mask word and count words are computed at compile time.

```
_optional ( expression1, expression2 )
```

Usage Guidelines

- The `_optional()` operator can be used only as an actual argument to a variable or extensible function.
- The first expression must be scalar.

Examples

1. This example shows a caller of an extensible function that dynamically decides to pass one of its arguments.

   ```
   _extensible foo (int, int, int);

   void bar (int i) {
      /* Pass a third argument of 30 to foo only if i is not zero. */
   ```

```
    foo (10, 20, _optional (i, 30));
}
```

2. This example shows a function that passes along one of its optional parameters only if an actual parameter is supplied.

```
_extensible foo (int, int, int);

_extensible void bar ( int i, int j) {
   /* Pass a third argument of j to foo only if j is present. */
   foo (10, 20, _optional (_arg_present (j), j ));
}
```

3. This example shows an incorrect usage of the _optional operator.

```
_extensible foo (int, int, int);
_extensible void bar (int i, int j) {
   foo (10,
        20,
        /* Error! The _optional operator is not used as an
           actual parameter.  It is used in an expression
           that is the actual parameter. */
        (_optional (_arg_present (j), j) + 1));
}
```

_typeof

The `_typeof` operator is the NonStop implementation of the GCC `typeof` extension to the C language. For more information on `typeof`, see **https://gcc.gnu.org/onlinedocs/gcc/Typeof.html**.

Data Types

The native C and C++ compilers, the TNS C compiler, and the TNS C++ preprocessor support the predefined data types described in the ISO/ANSI C standard. In addition, HPE has added three additional data types: *decimal,long long* and *unsigned long long*. **Predefined DATA Types** table summarizes the predefined data types.

Table 6: Predefined Data Types

Long forms	Abbreviated forms
`bool`[1]	`bool`
`char`	`char`
`short, signed short, short int,` or `signed short int`	`short`
`int, signed, signed int`	`int`
`long, signed long, long int,` or `signed long int`	`long`
`unsigned short` or `unsigned short int`	`unsigned short`
`unsigned` or `unsigned int`	`unsigned`
`unsigned long` or `unsigned long int`	`unsigned long`

Table Continued

Long forms	Abbreviated forms
`unsigned long long int`	`unsigned long long`
`long long int`	`long long`
`decimal`	`decimal`
`float`	`float`
`double`	`double`
`long double`	`long double`

[1] The bool type became available at the D45 release with native C++ using the VERSION2 directive.

decimal

The data type *decimal* is an HPE extension. *decimal* is applicable only for the native C and TNS C languages, which use the pragma SQL.

decimal specifies the predefined decimal type, which is used to access data in an SQL database. If you use *decimal*, you must specify the SQL pragma. If you want functions to manipulate *decimal* variables, you must specify the SQL pragma and include the header sqlh. For more details regarding the use of the data type *decimal*, see the *NonStop SQL/MP Programming Manual for C*.

long long

The data type *long long* is a 64-bit integer. This data type is applicable for the native and TNS C and C++ languages.

For TNS C++, type matching occurs for initialization, expressions, and argument lists. Variables of type *long long* are allowed in all contexts that TNS C allows them. Constants of type *long long* are recognized when the decimal number is followed by the modifier `LL` or `ll`. Variables and constants of type *long long* are allowed in classes and templates.

unsigned long long

The data type `unsigned long long` is an unsigned 64-bit integer. This is applicable for TNS/R, TNS/E, and TNS/X native C and C++ languages.

signed char

The data type `signed char` is an HPE extension to Cfront, the TNS C++ preprocessor. The data type `signed char` can be used to distinguish a specific instance of an overloaded member function.

Size of Type int

For native C and C++, the data type `int` is always 32 bits.

For TNS OSS programs, the data type `int` is always 32 bits. For TNS Guardian programs, the size of the data type int can be 16 bits or 32 bits, depending on whether you have specified the 32-bit (wid) data model. To specify the 32‑bit (wide) data model, use the WIDE pragma. The WIDE pragma compiles only under the large‑memory model.

For TNS C, a C program runs under the large-memory model unless you have specified the `NOXMEM` pragma.

For TNS C++, a C++ program always runs under the large-memory model. In addition, the 32-bit (wide) data model is the default specification. If you want the size of the data type `int` to be 16 bits, specify the `NOWIDE` pragma.

Relationship Between WIDE Pragma and Types `short`, `int`, and `long` table shows the size of the `short, int`, and `long` types when you omit or specify the `WIDE` pragma in TNS C.

Table 7: Relationship Between WIDE Pragma and Types `short`, `int`, and `long`

Data Type	Size Without WIDE Pragma Specified	Size With WIDE Pragma Specified
`short`	16 bits	16 bits
`int`	16 bits	32 bits
`long`	32 bits	32 bits

For TNS C and C++, relative sizes of the types short, int, and long depend on whether you compile the program under the 32-bit (wide) data model.

Without the 32-bit (wide) data model: With the 32-bit (wide) data model:

```
char < short = int < long    char < short < int = long
```

For application portability and compatibility with native C and C++, use the 32-bit (wide) data model whenever possible. For more details on the `WIDE` pragma, see **WIDE** on page 331. For more details on the 32-bit (wide) data model, see **Two Data Models: 16-Bit and ILP32** on page 434.

Interfacing to Guardian Procedures and OSS Functions

This chapter describes how to declare and call Guardian procedures and OSS functions in C and C++ programs.

Use Guardian procedures when you cannot accomplish a task with C functions. For example, to get the file code of a Guardian file, use a Guardian file-system procedure, such as FILE_GETINFO_ or FILE_GETINFOBYNAME_. (See also **Procedures With 16-Bit Addressable Parameters** on page 76.)

For more details about Guardian procedures, see the *Guardian Procedure Calls Reference Manual*. Some procedures in this manual are described as “superseded” by other procedures. For example, OPEN is superseded by FILE_OPEN_. Superseded procedures do not take full advantage of D-series or later features and you should not use them. They are provided for backward compatibility only.

Declaring Guardian Procedures

Like all functions in a C program, Guardian procedures must be declared before they can be called. Guardian procedures are declared as external procedures.

These library header files simplify the declaration of Guardian procedures:

System Type	Filename	Location
Guardian file system	cextdecs	`$SYSTEM.SYSTEM`
OSS	cextdecs.h	`/usr/include/sys`
PC	cextdecs.h	`$COMP_ROOT\usr\include`

These identical header files contain function prototype declarations for most of the Guardian procedures that you can call directly from C and C++ programs. These files also correspond to the TAL and pTAL EXTDECS header file.

Other header files in $SYSTEM.SYSTEM and `/usr/include` also contain Guardian procedure declarations. These header files declare Guardian procedures added in D40 or later releases and include the `setjmp`, `tdmsigh`, `dlaunchh`, and `histryh` header files.

To determine the header file that declares the procedure, see the procedure’s reference page in the *Guardian Procedure Calls Reference Manual*. Each procedure’s reference page includes its C declaration syntax. However, do not use this syntax to declare Guardian procedures. Include the appropriate header file instead.

The header files specify the C names for the Guardian procedures in uppercase characters and provide a section for each procedure using the `SECTION` pragma. The declarations provided by the header files follow the guidelines for declaring TAL procedures in **Mixed-Language Programming for TNS Programs** on page 113, and for declaring pTAL procedures in **Mixed-Language Programming for TNS/R, TNS/E, and TNS/X Native Programs** on page 141.

To access declarations in the header files, use the `#include` directive, specifying as section names the names of the Guardian procedures you want to include. This example includes the declarations of the PROCESS_GETINFOLIST_ and FILENAME_FINDNEXT_ procedures:

```
#include <cextdecs(PROCESS_GETINFOLIST_, FILENAME_FINDNEXT_)>
```

Many Guardian procedures are written in TAL or pTAL. The header file declarations use the _tal language-specifier to identify external routines. Native C and C++ programs that call Guardian procedures must specify the EXTENSIONS pragma, because _tal is an HPE extension.

Guardian procedures that use condition codes are declared as procedures returning _cc_status. Consequently, you must include the header talh (in the Guardian file system) or tal.h (in the OSS file system) before you call such a Guardian procedure.

The $SYSTEM.ZSYSDEFS.ZSYSC file contains a set of declarations, consisting mainly of named constants (literals) and data structure definitions, that can be used for parameters to Guardian procedures. Like the Guardian procedure header files, the ZSYSC file is divided into sections that allow you to include only those declarations that your program needs. This directive includes only the process-creation and system-message constant declarations:

```
#include "$system.zsysdefs.zsysc (process_constant,   \
                                  system_messages_constant"
```

Calling Guardian Procedures

The syntax of a Guardian procedure call determines whether a parameter is required or is optional. To call procedures with optional parameters, omit the parameter, but must include the comma that would follow it. For example:

```
err = FILENAME_SCAN_ (string,length,count,,,options);
```

A call to a Guardian procedure usually returns either a return value or a condition code. Most Guardian procedures fall into one of these two categories, and you can call these procedures directly from a C or C++ program. Those procedures that return both a return value and a condition code cannot be called directly from a C or C++ program.

Procedures That Return a Return Value

Calling a Guardian Procedure That Returns a Return Value on page 74 shows a C program that calls the Guardian procedure FILE_GETINFOBYNAME_ to get information about a Guardian file, which is provided as an input parameter to the procedure. The procedure returns a return value into retcode.

Note 1	The integer into which a value is returned, retcode, is declared as a short integer. This is because the integer data type in Guardian procedures represents a 16-bit word, while the integer data type in large-model and wide-model C and C++ programs represents a 32-bit word.
Note 2	You must insert a placeholder comma (,) if you omit optional parameters, unless you omit them from the end of the list. Omitted optional parameters contain default values, if default values are defined.

Calling a Guardian Procedure That Returns a Return Value

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cextdecs.h(FILE_GETINFOBYNAME_)>

char *filename;
short typeinfo[5];

int main(int argc, char *argv[]) {
   short retcode, physreclen;            /* Note 1 */

/* Get Guardian filename and pass it to the Guardian procedure. */
```

```
   if(argc > 1)
      filename = argv[1];
   else
      filename = "$system.system.stdioh";
   retcode = FILE_GETINFOBYNAME_(filename, /* Guardian filename */
      (short) strlen(filename),            /* filename length */
      typeinfo,              /* return array of file information */
      &physreclen,          /* returned physical record length */
      ,                     /* desired options, if any  Note 2 */
      ,                     /* tag or timeout value, if any    */
      );
/* Check for valid filename. */
   if (retcode != 0) {
     fprintf(stderr, "retcode = %d\n", retcode);
     fprintf(stderr, "Non-existent file or filename in bad format\n");
      exit(1);
   }

/* Print information for a disk file. */
   printf("Filename = %s\n", filename);
   printf("device type = %d\n", typeinfo[0]);
   printf("device subtype = %d\n", typeinfo[1]);
   printf("physical record length = %d\n", physreclen);
   if(typeinfo[0] == 3) {           /* it's a disk file */
      printf("object type = %d\n", typeinfo[2]);
      printf("file type = %d\n", typeinfo[3]);
      printf("file code = %d\n", typeinfo[4]);
   }
   return(0);
}
```

Procedures That Return a Condition Code

Calling a Guardian Procedure That Returns a Condition Code on page 75 calls the Guardian procedures FILE_OPEN_ and READX to open and read data from a Guardian file, which is provided as an input parameter to the procedure. FILE_OPEN_ returns a return value in `retcode`, and READX returns a condition code in `CC`.

Note 1	The `tal.h` library header file must be included when you call Guardian procedures that return condition codes.
Note 2	The `_cc_status` type specifier indicates that a procedure does not return a value but does set a condition code.

Calling a Guardian Procedure That Returns a Condition Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <tal.h>    /* Note 1 */
#include <cextdecs.h(FILE_OPEN_,READX)>

_cc_status CC;     /* Note  2 */
short filenum, bytesread, retcode;
char *filename;
char buffer[1024];
```

```
int main(int argc, char *argv[]) {
int i;

/* Get Guardian filename and open the file. */
   if (argc > 1)
      filename = argv[1];
   else
      filename = "$system.system.stdioh";
   retcode = FILE_OPEN_(filename,   /* name of Guardian file*/
          (short)strlen(filename),   /* filename length */
          &filenum,                  /* file number returned */
          1,                         /* open for read */
          ,                          /* exclusion */
          ,                          /* nowait depth */
          ,                          /* sync or receive depth */
          );
   /* Check if open was successful. */
    if (retcode != 0) {
       fprintf(stderr, "retcode = %d\n", retcode);
       fprintf(stderr, "Can't open %s\n", filename);
       exit(1);
    }

/* Read from the file and check for valid condition code. */
   CC = READX( filenum,          /* file number */
               buffer,           /* buffer address */
               1024,             /* size of buffer */
               &bytesread,       /* number of bytes read */
               );
   if(_status_ne(CC)) {    /* if condition code indicates an error */
      fprintf(stderr, "Read error, bytes read = %d\n", bytesread);
      exit(1);
      }
/* Print number of bytes read and contents of first ten bytes.*/
   printf("Bytes read = %d\n", bytesread);
   printf("First 10 bytes read (in octal): ");
   for (i = 0; i < 10; i++)
      printf("%o ", buffer[i]);
   printf("\n");
   return(0);
```

Procedures With 16-Bit Addressable Parameters

Native C and C++ programs do not support 16-bit addressable parameters. All Guardian procedures called from native programs use 32-bit addressable parameters. This subsection applies only to TNS C and C++ programs.

Some Guardian procedures require 16‑bit addressable parameters. 16‑bit addressable parameters are located in the user data segment instead of an extended data segment. Guardian procedures that require 16‑bit addressable parameters have corresponding versions that require 32‑bit addressable parameters. For example, READ requires 16-bit parameters and READX requires 32‑bit parameters. Use the Guardian procedures with 32‑bit addressable parameters in new code and consider converting old code as well.

Parameters located in the user data segment need to be declared and passed properly. Small-memory model TNS C programs only support 16-bit addressable data so no changes are required. Large-memory model and 32-bit (or wide) data model TNS C and C++ programs require you to use the NOXVAR pragma or the `_near` pointer modifier to denote aggregates that are to be stored in the user data segment. See **XVAR** on page 334 or **Attribute Specifier** on page 61.

Procedures That You Cannot Call

C and C++ programs can call most but not all Guardian procedures. For example, they cannot call the Guardian INITIALIZER and ARMTRAP procedures. To determine whether a Guardian procedure can be called from a C or C++ program, see the procedure's reference page in the *Guardian Procedure Calls Reference Manual*.

C and C++ programs cannot directly call Guardian procedures that both return a value and return a condition code. The subsections **TAL Procedures That You Cannot Directly Call** on page 123 and **pTAL Procedures That You Cannot Call Directly** on page 149 present techniques that enable you to access procedures in this category.

Declaring OSS Functions

Unlike Guardian procedures that are written in TAL or pTAL, Open System Services (OSS) functions are written in C. Therefore, to declare OSS functions, you need only to specify library header files that contain the prototype definitions for OSS functions called from your program. To determine the correct header files to specify, see the *Open System Services System Calls Manual*, the *Open System Services Library Calls Manual*, or the reference page for a particular function.

C and C++ programs that run in the Guardian environment cannot call all OSS functions. For more details, see the *Open System Services Programmer's Guide*.

Using the C Run-Time Library

Versions of the C Run-Time Library

The C run-time library has the following versions:

- Guardian TNS C run-time library
- Open System Services (OSS) G-series TNS C run-time library
- Guardian and OSS TNS/R native C run-time library
- Guardian and OSS TNS/E native C run-time library
- Guardian and OSS TNS/X native C run-time library

The library version used depends on the environment (NonStop) and mode (TNS or native) of a program.

Each library version provides the complete set of functions specified by the ISO/ANSI C standard. The ISO/ANSI C standard allows implementations to vary in specific instances. For more details, see **HPE C Implementation-Defined Behavior** on page 515.

The OSS TNS and OSS native library versions and the Guardian native library versions support additional functions specified by the XPG4 and XPG4 Version 2 specifications. For more details on standards compliance, see **Complying With Standards** on page 48.

Regardless of the data model of a Guardian module, you can call ISO C standard functions and HPE extension functions. From Guardian modules using the 32-bit (wide) data model, you can call many functions also specified in the XPG4 Version 1 or Version 2 specifications (X/OPEN UNIX95) or the POSIX standard. From Guardian modules using the 16-bit data model, you cannot call functions in the XPG4 specifications or the POSIX standard that are not also in the ISO C standard. (The ISO C standard is a subset of the XPG4 specifications and the POSIX standard.)

The C run-time libraries also include HPE extension functions for input and output, fault-tolerant programming, SQL data-type conversion, EDIT file manipulation, and process startup information retrieval.

Complete semantics and syntax of functions and macros in each version of the C run-time library are documented in these manuals:

Version	Manual
Guardian TNS C run-time library	*Guardian TNS C Library Calls Reference Manual*
OSS TNS C G-series run-time library	D30 edition of the *Open System Services Library Calls Reference Manual*
Guardian native C run-time libraries	*Guardian Native C Library Calls Reference Manual*
OSS native C run-time libraries	*Open System Services Library Calls Reference Manual*

Input/Output Models

The C run-time library provides direct input and output access to several types of physical files:

- C disk files, which are odd-unstructured files and have a file code of 180
- EDIT disk files, which have a file code of 101
- Processes
- Terminals

To provide uniform access to these various types of physical files, the C run-time library uses two logical file types, binary and text, to characterize and interpret the data found in a given file.

In addition, the C run-time library provides two sets of input and output functions. Each set has its own method of tracking, maintaining, and referring to a file. These methods, called file-reference models, are the ANSI model, which uses FILE pointers to identify files, and the alternate model, which uses file descriptors for this purpose. The ANSI model is the same for the Guardian and OSS environments. For TNS C, the alternate model is different between the Guardian and OSS environments. In the Guardian TNS environment, the alternate model is an HPE extension to the ISO/ANSI C standard. The OSS TNS environment, however, and in both the OSS and Guardian native environments use the model based on the XPG4 specification. (The Guardian native environment permits access to the alternate model.)

The discussion presented later in this section describes the alternate model defined by HPE for the Guardian TNS environment. For more details on the alternate model in both the OSS TNS and OSS native environments and in the Guardian native environment, see the reference pages for the alternate model I/O functions or an appropriate edition of the *Open System Services Library Calls Reference Manual*.

If no combination of file-reference model and logical file type provides the I/O services you require, call Guardian system procedures directly to perform specialized I/O functions. To find out how to access these procedures, see **Interfacing to Guardian Procedures and OSS Functions** on page 73.

Logical File Types

Regardless of the file-reference model you use, specify the logical file type of a physical file when you open the physical file. When selecting logical type to use, consider two factors:

1. The logical types best matches the way you want to interpret the data in the physical file
2. Logical type chosen in Step 1 is available for the given physical file type

The next two subsections describe how the two logical file types interpret and buffer data.

Binary-Type Logical Files

The C run-time library interprets a binary-type logical file as a sequence of arbitrary byte values. It does not translate data on input or output. Consequently, control the structure, content, and interpretation of a binary-type logical file.

Text-Type Logical Files

The C run-time library interprets a text-type logical file as a sequence of lines containing ASCII text. Terminating each line of text is a newline character `\n`. In addition, tab characters are converted to an appropriate number of spaces.

Both of the file-reference models perform line buffering of text files. However, both models enable you to disable this buffering. In the ANSI model, the `setnbuf()` function provides this service; in the alternate model, the `fcntl()` function does it.

Line buffering implies that data is buffered until a newline is transferred. However, the C run-time library also flushes the buffer when:

- the buffer becomes full. The maximum buffer sizes are listed in **Buffer Sizes** table.
- you explicitly flush the buffer (using `fflush()` or `fcntl()`).
- you make a call to `fclose()` or `exit()` .
- you make an input request of a nondisk file without flushing a series of output requests (by sending a newline or flushing the buffer).

Table 8: Buffer Sizes

File Type	Maximum Length of Buffer in Bytes
Terminals	239
EDIT files	239
C file accessed as text-type logical file	256
C file accessed as binary-type logical file	4096
Process accessed as text-type logical file	256
Process accessed as binary-type logical file	32,767
$RECEIVE	32,767

Physical File Types

As mentioned earlier, the C run-time library provides input and output access to several types of physical files, including C files, EDIT files, processes, and terminals. These subsections describe these physical file types and specify which logical types you can use to access each of the physical types.

C Files

C files are odd-unstructured disk files with a file code of 180 when stored in the Guardian file system. Because C files are odd-unstructured (rather than simply unstructured), they are byte-addressable; that is, you can read, write, and position to single bytes instead of words.

When accessing a C file, use either the text or the binary logical type.

Opening a C file as a text file causes subsequent writes and reads to transfer the exact number of characters specified, including trailing blanks.

Edit Files

When accessing an EDIT file, you must use the `text` logical type. Due to their physical structure, EDIT files are only line-addressable. Consequently, an error occurs if you attempt to disable line buffering of an EDIT file.

Because EDIT files do not physically contain newline characters, the run-time library automatically adds them on input and removes them on output. This addition of newlines makes the EDIT file truly accessible as a text file.

It is a standard feature of EDIT files that trailing blanks are stripped from each line.

Processes

When accessing a process, use either the binary or the text logical type.

When you characterize a process as a binary file, each I/O request is an interprocess message. Consequently, when you perform binary interprocess communication, you should use the direct I/O functions that transfer blocks of data, not single characters.

When you characterize a process as a text file, each line is an interprocess message. Because these messages do not physically contain newline characters, the run-time library automatically adds them on input and removes them on output.

Terminals

When accessing a terminal, use either the text or the binary logical type.

When you characterize a terminal as a text file, data is normally line-buffered. As a result, input from the terminal is not available until the terminal user presses the Return key.

When you characterize a terminal as a binary file, no buffering occurs.

If you open a terminal for alternate-model I/O as a binary file, an ambiguity arises when you read an empty line because the return value is zero (which normally indicates the end of the line). In this case, the alternate-model I/O functions set `errno` to the value 1 to indicate the end of the file.

$RECEIVE

When you are accessing $RECEIVE, HPE recommends that you use Guardian system procedures, rather than the ANSI or alternate I/O C functions, for:

- Performing interprocess communication
- Opening $RECEIVE and reading or replying to system messages

ANSI-Model I/O

As mentioned earlier, the ANSI-model I/O functions use FILE pointers to identify files. To create such a pointer, you make a declaration of the form:

```
FILE identifier;
```

The functions that open a physical file for ANSI-model I/O return a FILE pointer that denotes the newly opened file. From this point on, you use that FILE pointer to direct I/O requests to the physical file.

FILE pointers point to structures that are internal to the run-time library and contain various types of information regarding the file and its status. Several of the ANSI-model I/O functions offer access to various members of a FILE structure. These members include the file-position indicator and the file-error indicator.

Alternate-Model I/O

As mentioned earlier, the alternate-model I/O functions use file descriptors to identify files. The alternate-model I/O functions are not available in standard ISO/ANSI C. In the Guardian TNS environment, alternate-model I/O functions are defined by HPE. In all other environments, alternate-model I/O functions are defined by the XPG4 specification.

The arguments required by the function file descriptors are simply variables of type int, so to create one you make a declaration of the form:

```
int identifier;
```

The functions that open a physical file for alternate-model I/O return a file descriptor that denotes the newly opened file. You then use that file descriptor to direct I/O requests to the physical file.

Note that file descriptors are different from file numbers:

- Alternate-model I/O functions use file descriptors to identify files
- Guardian system procedures use file numbers to identify files

Use `errno` to determine whether there is an error when I/O is performed using the alternate-model I/O functions. The I/O error handling for the alternate model is an HPE extension not specified in the ISO/ ANSI C standard.

The header file `errnoh` or `errno.h` declares the variable `errno` and several object-like macros that are used to report errors. Include the `errnoh` header file, set `errno` to zero and call the alternate-model I/O function. If the function fails, `errno` contains the error code.

Mathematical Functions

This subsection describes mathematical functions available on HPE NonStop systems before the G06.06 release, and available after the G06.06 release if you specify Tandem floating-point format. Additional mathematical functions that are available with IEEE floating-point format are described in **IEEE Floating-Point Arithmetic** on page 82.

The C run-time libraries provide mathematical functions, such as `cos()` and `exp()`, defined in the ISO/ ANSI C standard and XPG4 specification. For specific conditions, these standards define the actions taken by a mathematical function. These actions are specified as either required or optional.

There are conditions defined in these standards that could not occur in the HPE implementation on older systems. These conditions include an argument or return value of NaN (not-a-number) and positive and negative infinity. Because these conditions could not occur, the Tandem floating-point format variants of the mathematical functions do not take actions defined in the standards.

For example, the XPG4 `cos()` function definition requires that NaN be returned if the parameter to the function is NaN. The definition further states that `errno` optionally might be set to `[EDOM]`. Because the Tandem floating-point format does not support NaN, the `cos()` function does not return NaN or optionally set `errno` to `[EDOM]`. Note that if you specify IEEE floating-point format, NaN is supported.

The standards specify the action that must or might occur for a given condition. If the condition cannot occur in a given implementation, the action is not required. Therefore, the HPE C run-time libraries conform to the standards. Many of the reference pages for the mathematical functions state the actions defined by the standards, even if the conditions that require those actions cannot occur in the HPE implementation. These actions are included to enable you to write code that has the highest level of portability to other environments.

IEEE Floating-Point Arithmetic

You have the option of choosing Tandem floating-point format or IEEE floating-point format for performing floating-point arithmetic in native C and C++ programs. Tandem floating-point format is the default for TNS/R systems; IEEE floating-point format is the default for TNS/E and TNS/X systems.

Tandem floating-point format is compatible with pre-G06.06 C and C++ applications. IEEE floating-point format, however, allows an application to take advantage of the greater performance provided by the floating-point instructions available in the processor hardware. Note that the results of IEEE floating-point operations can vary slightly between TNS/R, TNS/E, and TNS/X systems.

Differences Between Tandem and IEEE Floating-Point Formats

These are some high-level differences between the two floating-point formats:

- IEEE floating-point format covers a wider range of values for the `double` data type, but a narrower range of values for the `float` data type, than does Tandem floating-point format (see **Comparison of IEEE and Tandem Floating-Point Formats** table).
- IEEE floating-point format has greater precision than Tandem floating-point format.for the `float` data type, but slightly less precision for a `double`.
- IEEE floating-point format is faster than Tandem floating-point format.
- IEEE floating-point format is easier for porting applications.
- IEEE floating-point default handling of overflow, underflow, divide-by-zero, and invalid operation is better than the Tandem floating-point handling.
- IEEE floating-point directed roundings and “sticky” flags are useful debugging tools for investigating calculations that might go wrong because of rounding problems, division by zero, or other problems. Sticky flags are exception flags that stay set until explicitly reset by the user. They allow you to check a chain of computations.
- IEEE floating-point denormalized numbers avoid computational problems that arise from very small numbers as intermediate results in computations.
- IEEE floating-point format is available only on NonStop servers with the underlying hardware capacity, running a G06.06 or later version of the HPE NonStop OS.

For more general information about IEEE floating-point format, see **Compiling and Linking Floating-Point Programs** on page 386, and see the *Guardian Programmer’s Guide*. For more details about using IEEE floating-point format (including the exponent ranges and the number of bits of exponents and fractions), see the pragma **IEEE_FLOAT** on page 248.

Comparison of IEEE and Tandem Floating-Point Formats table lists the differences in precision between the two floating-point formats.

Table 9: Comparison of IEEE and Tandem Floating-Point Formats

Characteristic	IEEE float	Tandem float	IEEE double	Tandem double
Total bits in format	32	32	64	64
Bits of exponent	8	9	11	9
Bits of stored fraction	23	22	52	54
Nominal precision, including hidden bit	24	23	53	55

Active Backup Programming Functions

The term “fault tolerant” means that a single failure does not cause processing to stop. The C run‑time library enables you to write fault-tolerant programs using the active backup programming model. Active backup programs are supported only in the Guardian environment.

The active backup programming model is the only method to write fault-tolerant programs in C. Use of the Guardian CHECKMONITOR procedure to perform “passive” backup programming does not provide true fault tolerance in C programs.

In active backup programming, processes run in pairs: a primary process that performs the tasks of the underlying application, and a backup process that is ready to take over execution from the primary process if the primary process or processor fails. Active backup programs have these characteristics:

- Active backup uses process pairs to achieve fault tolerance.
- The primary process sends state information to the backup process. State information is information about the run-time environment that is required for the backup process to take over for the primary process.
- The backup process receives state information from the primary process, detects a failed primary process or CPU, and takes over execution.

An active backup program executes as a primary and backup process pair running the same program file. The primary and backup processes perform interprocess communication. The primary process sends critical data to the backup process. This critical data serves two purposes: to provide sufficient information to enable the backup process to resume application processing (file state and application state information), and to indicate to the backup process where it should logically resume application processing (control state information).

The backup process receives messages from two sources. It receives critical information from the primary process (state information), which it must record for future use in the event it must take over processing from the primary process. It can also receive messages from the operating system indicating that the primary process or processor has failed. If the primary process fails, the backup process takes over processing at the logical point in the application indicated by the most recent control state information received from the primary process, and it continues processing using the most recent file state and application state information.

C Functions for Active Backup Programming table lists the functions used to write active backup programs.

Table 10: C Functions for Active Backup Programming

Function	Use
`__ns_start_backup()`	Called by the primary process to create and initialize the backup process.
`__ns_backup_fopen()`	Called by the backup process to open files that have already been opened by the primary process.
`__ns_fget_file_state()`	Called by the primary process to obtain file state information.
`__ns_fset_file_state()`	Called by the backup process to update file state information.
`__ns_fget_file_open_state()`	Called by the primary process to obtain open state information for a file.
`__ns_fopen_special()`	Called by the primary process to open a file with a specified sync depth.

These functions are described in the *Guardian TNS C Library Calls Reference Manual* and *Guardian Native C Library Calls Reference Manual*. See the *Guardian Programmer’s Guide* for a description of active backup programming, including:

- An overview of the activities an active backup program must perform
- Detailed explanations of how to code an active backup program
- Examples of active backup programs

Environment-Specific Functions

There are several file-system functions that allow you to operate on a file in the opposite environment without writing and compiling a separate module for that environment. These functions are `fopen()`, `freopen()`, `remove()`, `rename()`, `tmpfile()`, and `tmpnam()`. Each of these functions has environment-specific variants with suffixes to indicate which type of file you want to operate on. For example, `fopen_guardian()` opens a Guardian file in an OSS module and `fopen_oss()` opens an OSS file in a Guardian module.

NOTE:

If a TNS module calls any of the environment-specific variants, the feature-test macro `_INTEROPERABLE` must be specified when the module is compiled. Specifying `_INTEROPERABLE` allows subsequent I/O function calls to know which type of I/O stream they are operating on.

For more details, see the *Open System Services Programmer's Guide*.

Changes Required to Interoperable Compilation Modules at D44

At the D44 release, preexisting TNS/R native C programs that call any of these functions to access files in the OSS environment might need to be recompiled with a native C compiler:

```
fopen()
fopen_oss()
freopen()
freopen_oss()
remove()
remove_oss()
rename()
rename_oss()
tmpnam()
tmpnam_oss()
tmpfile()
tmpfile_oss()
```

After the D44 release, each of these functions has a Guardian variant, such as `fopen_guardian()`, and each has an OSS variant, such as `fopen_oss()`.

Recompilation is required for compilations made using a compiler released before D44 (G04) if the execution environment is to be D44 or later (G04 or later).

You do not need to recompile Guardian TNS/R native programs that call these functions to access Guardian files, and OSS programs that call these functions to access OSS files.

The TNS/R native C modules that must be recompiled and the source code changes you need to make are summarized in this table:

Type of Compilation Unit	Type of Executable Unit	Change Required
`SYSTYPE GUARDIAN`	OSS process	Use `_guardian` variant of the function in compilation unit.
`SYSTYPE OSS`	Guardian process	Use `_oss` variant of the function in compilation unit.

EDIT File Functions

The Guardian TNS and native C run-time libraries provide two HPE extension functions to support EDIT files: `edfseek()` and `edftell()`. For more details, see the *Guardian TNS C Library Calls Reference Manual*, *Guardian Native C Library Calls Reference Manual,* or *Open System Services Library Calls Reference Manual.*

SQL Data-Type Conversion Functions

The Guardian TNS and native C run-time libraries provide HPE extension functions to support data-type conversion for SQL programs. The two data-type conversion functions are `dec_to_longlong()` and `longlong_to_dec()`. For more details, see the *Guardian TNS C Library Calls Reference Manual*, *Guardian Native C Library Calls Reference Manual,* or *Open System Services Library Calls Reference Manual*.

Startup Information Retrieval Functions

The Guardian TNS and native C run-time libraries include six functions that allow the retrieval of the process startup message, the PARAM message, and the ASSIGN messages. You can call these functions from Guardian processes only. These functions are listed in the **C Functions That Retrieve Process Startup Information** table. For a detailed description of each function, see the *Guardian TNS C Library Calls Reference Manual* or *Guardian Native C Library Calls Reference Manual*.

Table 11: C Functions That Retrieve Process Startup Information

Function	Action
`get_assign_msg()`	Retrieves a specified ASSIGN message.
`get_assign_msg_by_name()`	Retrieves the ASSIGN message that corresponds to the logical-unit name requested.
`get_max_assign_msg_ordinal()`	Determines how many ASSIGN messages are assigned to a particular process.
`get_param_by_name()`	Retrieves the value of the parameter that corresponds to the parameter name requested.
`get_param_msg()`	Retrieves the PARAM message.
`get_startup_msg()`	Retrieves the process startup message.

Miscellaneous HPE Extension Functions

These functions are HPE extensions that provide specific control over the HPE C and C++ environment on NonStop systems:

`chvol()`

`heap_check()`

`heap_check_always()`

`heap_min_block_size()`

`fopen_std_file()`

`terminate_program()`

For more details, see the *Guardian TNS C Library Calls Reference Manual*, *Guardian Native C Library Calls Reference Manual,* or *Open System Services Library Calls Reference Manual*.

These functions are HPE extensions that a 64-bit data model process can use to access a secondary 32-bit addressable heap space:

`calloc32()`

`free32()`

`malloc32()`

`realloc32()`

`heap_check32()`

`heap_check_always32()`

For more details, see the *Open System Services Library Calls Reference Manual*.

Using the Standard C++ Library

Four versions of the standard C++ library are available as listed in the following table.

VERSION3 is the default native C++ run-time library. For more details about the default language dialect, see **VERSION3 (page 262)**. For information about all available libraries, see **Table 44: SRLs Available When Using VERSION1, VERSION2, and VERSION3** and **Table 50: DLLs Available When Using VERSION2, VERSION3, and VERSION4**.

Table 12: Versions of the Standard C++ and C++ Run-Time Libraries

Version	Product Number	Description
TNS/R programs		
VERSION1	T9227	C++ run-time library
NonStop S-series systems only		Separate libraries for Guardian and OSS: ZCPLGSRL and ZCPLOSRL respectively
VERSION2	T2824	Common header files for all C++ versions
	T5895	Standard C++ library, Rogue Wave Software version 1.31, file ZRWSLSRL
	T0179	C++ run-time library, file ZCPLSRL
	T8473	Tools.h++ version 7 library, file ZTLHSRL
	T5894	Tools.h++ version 7 header files
	T2824	Common header files for all C++ versions
VERSION3	T2767	Standard C++ library ISO/IEC (Dinkumware) combined with the C++ run-time library, file ZSTLSRL
	T2824	Common header files for all C++ versions
TNS/E programs		
VERSION2	T2832	VERSION2 C++ standard library that contains classes, data, and functions that are not shareable with their VERSION3 counterpart or do not exist in VERSION3, file ZCPP2DLL.
	T2834	Tools.h++ version 7 header files
	T2835	Tools.h++ version 7 library, file ZTLH7DLL
	T2831	Common library for VERSION2 and VERSION3, file ZCPPCDLL.
	T2830	Common header files for all C++ versions

Table Continued

Version	Product Number	Description
VERSION3	T2831	Common library for VERSION2 and VERSION3, files ZCPPCDLL and YCPPCDLL.
	T2833	VERSION3 C++ standard library that contains classes, data, and functions that are not shareable with their VERSION2 counterpart or do not exist in, files ZCPP3DLL and YCPP3DLLVERSION2.
	T2830	Common header files for all C++ versions
TNS/X programs		
VERSION2	T2832	VERSION2 C++ standard library that contains classes, data, and functions that are not shareable with their VERSION3 counterpart or do not exist in, file XCPP2DLL VERSION3.
	T2834	Tools.h++ version 7 header files
	T2835	Tools.h++ version 7 library, file XTLH7DLL
	T2831	Common library for VERSION2, VERSION3, VERSION4, and file XCPPCDLL.
	T2830	Common header files for all C++ versions
VERSION3	T2831	Common library for VERSION2, VERSION3, and VERSION4, files XCPPCDLL and WCPPCDLL.
	T2833	VERSION3 C++ standard library that contains classes, data, and functions that are not shareable with their VERSION2 counterpart or do not exist in VERSION2, files XCPP3DLL and WCPP3DLL.
	T2830	Common header files for all C++ versions
VERSION4	T2830	Common header files for all C++ versions
	T2831	Common library for VERSION2, VERSION3, and VERSION4, files XCPPCDLL and WCPPCDLL.
	T2837	VERSION4 C++ standard library that contains classes, data, and functions for c++11 support. This library can only be used with VERSION4 compiles. Files are XCPP4DLL and WCPP4DLL.

CAUTION: All modules of an application must be built using the same version of the standard C++ library dialect. The native linkers and the NonStop OS perform version checking. Attempting to mix versions will yield an error or a warning at link time, and a run-time error at load time.

User Documentation

Available in the NonStop Technical Library (NTL):

- VERSION3:

  Standard C++ Library Reference ISO/IEC (VERSION3)

  > ⚠ **CAUTION:** The VERSION3 documentation contains descriptions of an underlying C library that is not the same in every case as the C library supported on HPE NonStop systems. The descriptions of those C functions might not apply to the NonStop C run-time library.

- VERSION2:

  Standard C++ Library User Guide and Tutorial and the Standard C++ Library Class Reference

- VERSION1:

  AT&T C++ Reference, Release 3 or The Annotated C++ Reference Manual by Margaret Ellis and Bjorne Stroustrup (these books are not available on NTL).

Features of the Standard C++ Library (VERSION3)

VERSION3 Feature	Advantages
C++ Standard Compliant	The language and library provide full compliance with the ANSI/ISO C++ Standard, with very few exceptions. ANSI/ISO compliance simplifies porting applications to the NonStop platform.
Improved exception handling	Type checking for throws and catches has been improved and is tighter than for VERSION2. Type IDs (names beginning with `__TID_`) are used to match throws and catches. The "const-ness" of the catch must match that of the throw. For example, if "goodbye world" is thrown, the catch must expect a `const char *`.
Template support	Templates are now fully supported, including templates as parameters to templates, and explicit qualification of templates.
Support for DLLs	Beginning with G06.20, both VERSION2 and VERSION3 libraries fully support using dynamic-link libraries (DLLs). You must use the latest libraries in a DLL environment; VERSION1 does not support the use of DLLs.
Floating point	Both IEEE_FLOAT and TANDEM_FLOAT are supported by VERSION2 and VERSION3. All object files that make up a running process should be consistent in the use of floating-point type. VERSION1 does not support IEEE_FLOAT.
Pthreads	To ensure that the same thread is used to throw and catch an exception, Pthreads and the C++ VERSION2 or VERSION3 run-time libraries store some state information in variables in the C++ library. This functionality has been implemented in both VERSION3 and VERSION2.

Table Continued

VERSION3 Feature	Advantages
Migration tool	To help customers move from C++ VERSION2 to C++ VERSION3, the migration tool generates warnings in the listing file where potential problems are found. The MIGRATION_CHECK pragma is described in **MIGRATION_CHECK** on page 268 , and its output is described in **MIGRATION_CHECK Messages** on page 569 .
Hash tables	Although not part of the standard, HASH_SET, HAS_MULTISET, HASH_MAP, and HASH_MULTIMAP have been implemented in the HPE NonStop C++ implementation.
Support for the neutral C++ dialect on TNS/E systems	Although not part of the standard, HPE provides the features described in **Using the Neutral C++ Dialect** on page 102 . These features take advantage of the repackaging of C++ objects on TNS/E systems into three library files so that a limited ability exists to create libraries that are not version dependent.

Contents of the Standard C++ Library

VERSION4

For `VERSION4`, the Standard C++ Library enforces the ISO/IEC IS 14882:2011 standard for C++ and a number of fixes ratified by the C++ Standards Committee. The C++ standard and fixes are documented in corrigenda before the NonStop `VERSION4` code was developed and finalized.

For `VERSION4`, the c++11 Standard Library (ISO/IEC14882:2011) is a port of the LLVM libc++. The NonStop implementation is based on LLVM version 3.7.0. For Information about the Standard c++ library, see the descriptions of the C++ library headers that declare or define library entities for the program.

The `VERSION4` Standard C++ Library contains all the headers included in the `VERSION3` library and the following headers:

```
<array>
<chrono>
<codecvt>
<conditional_variable>
<forward_list>
<initializer_list>
<random>
<regex>
<scoped_allocators>
<system_error>
<tuple>
<type_traits>
<typeindex>
<unordered_map>
<unordered_set>
<ccomplex>
<cfenv>
<cinttypes>
<cstdalign>
<cstdbool>
```

<cstdint>

<ctgmath>

Using VERSION4 on the Guardian environment, you can use the full name of the standard headers, without truncating the name or using a .h file extension. The NonStop system automatically performs truncation as necessary. HPE recommends that you also use the CPATHEQ pragma to specify the SLMAP file as described in **Pragmas for the Standard C++ Library**.SLMAP contains specific truncation rules for the header names that do not fit the standard.

VERSION3

For VERSION3, the Standard C++ Library ANSI/ISO/IEC is a port of the Dinkumware C++ Library, a conforming implementation of the Standard C++ library by P.J. Plauger. The library enforces the ISO/IEC IS 14882:1998(E) standard for C++ and a number of fixes ratified by the C++ Standards Committee. The C++ standard is documented in corrigenda before the NonStop VERSION3 code was developed and finalized.

A C++ program can call many functions from the Dinkum C++ Library that performs essential services such as input and output. These functions also provide efficient implementations of frequently used operations. Numerous function and class definitions accompany these functions to help you make better use of the library.

Information about the Standard C++ library can be found in the descriptions of the C++ library headers that declare or define library entities for the program. Also see the documentation for the library, the *Standard Compliant C++ Library Reference*, available on the NonStop user CD using the HPE NonStop Technical Library (NTL).

There are two broad subdivisions of the Standard C++ headers: streams and the standard template library (STL).

The VERSION3 Standard C++ Library contains the following C++ headers:

<algorithm>

<bitset>

<complex>

<deque>

<exception>

<fstream>

<functional>

<hash_map>

<hash_set>

<iomanip>

<ios>

<iosfwd>

<iostream>

<istream>

<iterator>

<limits>

<list>

<locale>

<map>

<memory>

<new>

<numeric>

<ostream>

<queue>

```
<set>
<slist>
<sstream>
<stack>
<stdexcept>
<streambuf>
<string>
<strstream>
<typeinfo>
<utility>
<valarray>
<vector>
```

Using `VERSION3` on Guardian environment, you can use the full name of the standard headers, without truncating the name or using a `.h` file extension. The NonStop system automatically performs truncation as necessary.

HPE recommends that you also use the `CPATHEQ` pragma to specify the SLMAP file as described in **Pragmas for the Standard C++ Library (page 89)**. SLMAP contains specific truncation rules for the header names that do not fit the standard.

These 18 C-name headers are also part of the library, required as part of the ANSI / ISO C++ Standard:

```
<cassert>
<cctype>
<cerrno>
<cfloat>
<ciso646>
<climits>
<clocale>
<cmath>
<csetjmp>
<csignal>
<cstdarg>
<cstddef>
<cstdio>
<cstdlib>
<cstring>
<ctime>
<cwchar>
<cwctype>
```

The C-name headers include the "*name.h*" header files that are part of the C run-time library and put the contents into the `std` namespace.

VERSION2

When `VERSION2` is specified, the Standard C++ Library from Rogue Wave Software is available. This library includes data structure and algorithm classes, plus string and numeric limits, complex classes, and allocators.

The `VERSION2` C++ Library includes:

- A large set of data structures and algorithms formerly known as the Standard Template Library (STL)
- A locale facility

- A templatized `string` class
- A templatized `complex` class for representing complex numbers
- A uniform framework for describing the execution environment through the use of a template class named `numeric_limits` and specializations for each fundamental data type
- Memory management features
- Language support features
- Exception handling features

VERSION1

`VERSION1` of the C++ Library includes the data structures and algorithm libraries and the `string`, `complex`, and `numeric_limits` classes. `VERSION1` does not include templates or exception handling.

`VERSION1` is not supported on TNS/E or TNS/X systems. To run a C++ program on a TNS/E or a TNS/X system when it is based on `VERSION1` of the C++ Library, you must migrate it to `VERSION3, VERSION2`, or `CPPNEUTRAL` before recompiling it.

Migration from `VERSION1` to `VERSION2` also requires migrating from use of Tools.h++ 6.1 to use of Tools.h++ 7. Migration from `VERSION1` to `VERSION3` requires migrating from use of either version of Tools.h++. See **Using Tools.h++** on page 104 and **Migration Considerations for Version 7** on page 105.

Installation Notes for VERSION4

The standard C++ Library ISO/IEC is delivered on the site update tape (SUT) for a NonStop system. On PC, the standard C++ Library is installed from a DVD that contains the Native C/C++ cross compiler.

The following table provides the location of the `Version4` standard C++ library installation in the Guardian and OSS environments.

Table 13: Installation Details for Standard C++ Library ISO/IEC (`VERSION4`)

Environment	Location of Headers	Location of Libraries
Guardian	`$SYSTEM.SYSTEM`	`$SYSTEM.ZDLL`*nnn*`.XCPP4DLL` and `$SYSTEM.ZDLL`*nnn*`.WCPP4DLL` on systems running L-series RVUs
OSS	`/usr/include`	The Guardian namespace: `$SYSTEM.ZDLL`*nnn*`.XCPP4DLL` and `$SYSTEM.ZDLL`*nnn*`.WCPP4DLL` on systems running L-series RVUs

Using Header Files With VERSION4

Specify the names of the `VERSION4` Standard C++ Library header files as defined in the standard. On NonStop systems, header files are renamed in some cases to implement the three versions of the library, but in all cases, you need to specify the standard names.

To use a `VERSION4` header:

you can specify:

```
#include <new>
```

But you cannot specify:

```
#include <new.h>
```

Logic is built into the header files to redirect your calls to the version of the library you are using. For example, if you are using `VERSION2` and you specify:

```
#include <exception>
```

Logic in the header files redirects you to the `VERSION2` header file, which is actually named `EXCEPTION2` (OSS and Windows) or `EXCEPTI2` (Guardian). You can see these alternate header file names in your program listing, but your `include` command should specify only the standard name, such as `exception` in this example.

Installation Notes for VERSION3

The Standard C++ Library ISO/IEC is delivered on the site update tape (SUT) for a NonStop system. On the PC, the Standard C++ Library is installed from a CD that contains the Native C/C++ cross compiler.

Installation Details for Standard C++ Library ISO/IEC (VERSION3) table summarizes where the parts of the `VERSION3` Standard C++ Library are installed in the different available environments (Guardian environment, OSS environment, and PC running Windows).

Table 14: Installation Details for Standard C++ Library ISO/IEC (VERSION3)

Environment	Location of Headers	Location of Libraries
Guardian	$SYSTEM.SYSTEM	$SYSTEM.SYS*nn*.ZSTLSRL on systems running G-series RVUs $SYSTEM.ZDLL*nnn*.ZCPP3DLL on systems running H-series or J-series RVUs $SYSTEM.ZDLL*nnn*.XCPP3DLL on systems running L-series RVUs
OSS	`/usr/include`	The Guardian namespace: `$SYSTEM.SYS`*`nn`*`.ZSTLSRL` on systems running G-series RVUs `$SYSTEM.ZDLL`*`nnn`*`.ZCPP3DLL` on systems running H-series or J-series RVUs `$SYSTEM.ZDLL`*`nnn`*`.XCPP3DLL` on systems running L-series RVUs

Using Header Files With VERSION3

Specify the names of the `VERSION3` Standard C++ Library header files as defined in the standard. On NonStop systems, header files have been renamed in some cases to implement the three versions of the library, but in all cases you need only to specify the standard names.

To use a `VERSION3` header file, do not include `.h` (even if the standard name contains `.h`).

For example, you can specify:

```
#include <new>
```

But you cannot specify:

```
#include <new.h>
```

Logic has been built into the header files to redirect your calls to the version of the library you are using. For example, if you are using `VERSION2` and you specify:

```
#include <exception>
```

Logic in the header files redirects you to the `VERSION2` header file, which is actually named `EXCEPTION2` (OSS and Windows) or `EXCEPTI2` (Guardian). You can see these alternate header file names in your program listing, but your `include` command should specify only the standard name, such as `exception` in this example.

Installation Notes for VERSION2

The Standard C++ Library is delivered on the site update tape (SUT) for a NonStop system. On the PC, the Standard C++ Library is installed from a CD that contains the Native C/C++ cross compiler.

Installation Details for Rogue Wave Standard C++ Library (VERSION2) summarizes where the parts of the `VERSION2` Standard C++ Library are installed in the different environments (Guardian environment, OSS environment, and PC running Windows).

Table 15: Installation Details for Rogue Wave Standard C++ Library (VERSION2)

Environment	Location of Headers	Location of Libraries
Guardian	$SYSTEM.SYSTEM	TNS/R code: $SYSTEM.SYS*nn*.ZRWSLSRL TNS/E code: $SYSTEM.ZDLL*nnn*.ZRWSLDLL TNS/X code: $SYSTEM.ZDLL*nnn*.XRWSLDLL
OSS	/usr/include	The Guardian namespace: TNS/R code: $SYSTEM.SYS*nn*.ZRWSLSRL TNS/E code: $SYSTEM.ZDLL*nnn*.ZRWSLDLL TNS/X code: $SYSTEM.ZDLL*nnn*.XRWSLDLL
PC running Windows	C:\tdmxdev\rel\includewhere *rel* is the release identifier, such as `d45`	C:\tdmxdev\rel\libwhere *rel* is the release identifier, such as `d45`

Several Standard C++ Library header files have been renamed in the `VERSION2` implementation for TNS/R-targeted compilations (the Guardian names, inside parentheses, are derived from the first seven letters plus the last letter of the header file name):

Documented Name	Name in VERSION2 Standard C++ Library (TNS/R-targeted Compilations only)
`exception` (EXCEPTIN)	`rwexcept` (RWEXCEPT)
`new` (NEW)	`rwnew` (RWNEW)
`stdexcept` (STDEXCET)	`rwstdex` (RWSTDEX)

This renaming avoids conflicts between the C++ run-time header files and the Rogue Wave Standard C++ Library header files.

If you are using the Rogue Wave Standard C++ Library (`VERSION2`), for an application targeted for TNS/R systems, specify the header files listed here that begin with RW (that is, `RWEXCEPT` instead of `EXCEPTION`, `RWNEW` instead of `NEW`, and `RWSTDEX` instead of `STDEXCEPT`). If you are using this library for TNS/E-targeted compilations, use the standard names. Logic has been built into the header files to redirect your calls to the version of the library you are using.

Examples of VERSION2 Headers

NOTE: Regardless of the version of the library you are using, the standard header files in the G06.23 or subsequent RVU automatically redirect calls to the appropriate header file.

1. Specifying the header-file name `new` includes `new.h` from the C++ run-time library in the HPE implementation (not the header `new` in the Standard C++ Library). **See: example**
2. Specifying the header-file name `rwnew` includes both `new.h` and the header file `rwnew` from the Standard C++ Library (as renamed for the HPE implementation).

   ```
   #include <rwnew>
   //includes standard header "new.h” in addition to “new"
   (renamed rwnew) from //Standard C++ Lib (T5895)
   ```
3. Specifying the header-file name `stdexcept` includes `stdexcept` in the C++ run-time library in the HPE NonStop implementation (not the header file `stdexcept`, renamed `rwstdex`, in the Standard C++ Library).

   ```
   #include <stdexcept>
   //includes header "stdexcept" from C++ RTL V2 (T0179)
   ```
4. Specifying the header-file name `rwstdex` includes the header file `stdexcept` from the Standard C++ Library (as renamed for the HPE implementation).

   ```
   #include <rwstdex>
   //includes header "stdexcept" from Standard C++ Lib V2 (T5895)
   ```

```
#include <new>
//includes header "new,h" from C++ Run-time Library V2 (T0179)
```

VERSION2 Standard C++ Library Example Files

Example files provided with the VERSION2 Standard C++ Library are located in $SYSTEM.ZLIBCPL on the NonStop system.

Table 16: VERSION2 Standard C++ Library Example Files

Example File	Description of Contents
MAKEEXAM	MAKE file for the examples. It is an OBEY command file.
CPLUS	Macro used by MAKEEXAM to build each test. You need to modify this file to reflect your own environment.
LINKIT	Contains instructions to nld for linking each test. This file is used by CPLUS. You need to modify this file to reflect your own environment.
ACCUMC	Manual Example Source File
ADJDIFFC	Manual Example Source File
ADVANCEC	Manual Example Source File
AUTOPTRC	Manual Example Source File
BINDERSC	Manual Example Source File
BITTESTC	Manual Example Source File
BSEARCHC	Manual Example Source File
COMTESTC	Manual Example Source File
COPYEXC	Manual Example Source File
COUNTC	Manual Example Source File
DEQUEC	Manual Example Source File
DISTANCC	Manual Example Source File
EQLRANGC	Manual Example Source File
EQUALC	Manual Example Source File
EXCEPTC	Manual Example Source File
FILLC	Manual Example Source File
FINDC	Manual Example Source File
FINDENDC	Manual Example Source File
FINDFOC	Manual Example Source File
FOREACHC	Manual Example Source File

Table Continued

Example File	Description of Contents
FUNCTOBC	Manual Example Source File
GENERATC	Manual Example Source File
HEAPOPSC	Manual Example Source File
INCLUDEC	Manual Example Source File
INRPRODC	Manual Example Source File
INSITRC	Manual Example Source File
IOITERC	Manual Example Source File
LEXCOMPC	Manual Example Source File
LIMTESTC	Manual Example Source File
LISTC	Manual Example Source File
MAPC	Manual Example Source File
MAXC	Manual Example Source File
MAXELEMC	Manual Example Source File
MERGEC	Manual Example Source File
MISMATCC	Manual Example Source File
MULTIMAC	Manual Example Source File
MULTISEC	Manual Example Source File
NEGATORC	Manual Example Source File
NTHELEMC	Manual Example Source File
PQUEUEC	Manual Example Source File
PARTSORC	Manual Example Source File
PARTSUMC	Manual Example Source File
PERMUTEC	Manual Example Source File
PNT2FNCC	Manual Example Source File
PRTITIOC	Manual Example Source File

Table Continued

Example File	Description of Contents
QUEUEC	Manual Example Source File
REMOVEC	Manual Example Source File
REPLACEC	Manual Example Source File
REVITRC	Manual Example Source File
REVERSEC	Manual Example Source File
RNDSHUFC	Manual Example Source File
ROTATEC	Manual Example Source File
SEARCHC	Manual Example Source File
SETDIFFC	Manual Example Source File
SETINTRC	Manual Example Source File
SETSDIC	Manual Example Source File
SETUNINC	Manual Example Source File
SETEXC	Manual Example Source File
SORTC	Manual Example Source File
STACKC	Manual Example Source File
STRTESTC	Manual Example Source File
SWAPC	Manual Example Source File
TRNSFORC	Manual Example Source File
ULBOUNDC	Manual Example Source File
UNIQUEC	Manual Example Source File
VECTORC	Manual Example Source File
ALG1C	Tutorial Example Source File
ALG2C	Tutorial Example Source File
ALG3C	Tutorial Example Source File
ALG4C	Tutorial Example Source File

Table Continued

Example File	Description of Contents
ALG5C	Tutorial Example Source File
ALG6C	Tutorial Example Source File
ALG7C	Tutorial Example Source File
AUTTESTC	Tutorial Example Source File
CALCC	Tutorial Example Source File
COMPLXC	Tutorial Example Source File
CONCORDC	Tutorial Example Source File
EXCEPTNC	Tutorial Example Source File
ICECREAC	Tutorial Example Source File
RADIXC	Tutorial Example Source File
SIEVEC	Tutorial Example Source File
SPELLC	Tutorial Example Source File
TELEC	Tutorial Example Source File
WIDWORKC	Tutorial Example Source File

Compiling and Linking in the OSS Environment

In addition to the C++ compiler pragmas recognized as command-line flags as described in **Compiler Pragmas** on page 193, the OSS `c89` and `c99` utilities have a `-Wcplusplus` flag that affects both compilation and linking. This flag:

- Defines the `__CPLUSPLUS` feature-test macro to modify C header files for C++.
- Causes all specified files with names suffixed by `.c` to be compiled as C++ files.

If a linker is also involved or in other words if `-c` flag is not used, using the `-Wcplusplus`flag causes the standard C++ libraries to be linked.

If the `-Wcplusplus` flag is not specified and none of the files operated upon by `c89` or `c99` has a suffix of `.C`, `.cpp`, `.cc`, or `.cxx`, `c89` or `c99` compiles the source files only as C files; if the `-c` flag is also omitted, `c89` or `c99` only links the C standard library.

Pragmas for the Standard C++ Library

You need to use the `MAPINCLUDE` and `CPATHEQ` pragmas to map UNIX or OSS pathnames to Guardian file names. HPE provides a CPATHEQ file for this purpose; the file is named SLMAP for the Standard C+ + Library. Like any CPATHEQ file, the SLMAP file maps OSS directory and file names to the Guardian namespace (*$volume.subvolume*). Similarly, there is a TLHMAP file for mapping names in Tools.h++ version 7.

The contents of a CPATHEQ file such as the SLMAP file:

```
#ifndef __SLMAP
#define __SLMAP
#pragma mapinclude      "rw/math.h"    = "mathh"
#pragma mapinclude      "rw/random.h"  = "randomh"
#pragma mapinclude      "rw/stddefs.h" = "stddefsh"
#pragma mapinclude      "rw/stdgen.h"  = "stdgenh"
#pragma mapinclude file "algorithm.cc" = "algoricc"
#pragma mapinclude file "bitset.cc"    = "bitsetcc"
#pragma mapinclude file "complex.cc"   = "complecc"
#pragma mapinclude file "deque.cc"     = "dequecc"
#pragma mapinclude file "iterator.cc"  = "iteratcc"
#pragma mapinclude file "list.cc"      = "listcc"
#pragma mapinclude file "string.cc"    = "stringcc"
#pragma mapinclude file "tree.cc"      = "treecc"
#pragma mapinclude file "vector.cc"    = "vectorcc"
#pragma mapinclude file "sys/types.h"  = "systypeh"
#pragma mapinclude file "sys/stat.h"   = "sysstath"
#pragma mapinclude file "hash_map"     = "hashmap"
#pragma mapinclude file "hash_set"     = "hashset"
#endif /* __SLMAP */
```

Note that this SLMAP file can be used with both `VERSION2` and `VERSION3`.

To use the SLMAP file, enter a CPATHEQ pragma specifying the location of the SLMAP file. For example:

```
#pragma cpatheq "$system.system.slmap"
```

For more details, including examples, about using the MAPINCLUDE and CPATHEQ pragmas with the Rogue Wave Software products (Tools.h++ and the Standard C++ Library), see **Pragmas for Tools.h++** on page 108. Also see the pragmas **CPATHEQ** on page 216 and **MAPINCLUDE** on page 266.

Using the Neutral C++ Dialect

A C++ dialect called the neutral C++ dialect, is defined for the TNS/E native environment. This dialect, called `CPPNEUTRAL`, consists of library components that are common to both `VERSION2` and `VERSION3` of the Standard C++ Library.

The neutral C++ dialect is used on TNS/E systems by system and middleware libraries so that C++ programs using either the `VERSION2` or `VERSION3` Standard C++ Library can use the system and middleware libraries. You can create your own DLL using the neutral C++ dialect so that your DLL can be similarly used by either a `VERSION2` or `VERSION3` C++ program.

Two pragmas support the neutral C++ dialect:

- `NEUTRAL`
- `BUILD_NEUTRAL_LIBRARY`.

The `NEUTRAL` pragma is only allowed in C++ standard headers. This pragma marks a class definition as being sharable between `VERSION2` and `VERSION3` of the Standard C++ Library. If a class definition that is not marked as such is used within a user-declared type or function that is marked by either the `export$` or `import$` specifier, the compiler can be set to issue an error message that indicates a version-dependent interface is used in a DLL.

The `BUILD_NEUTRAL_LIBRARY` pragma enables you to create a DLL that uses the neutral dialect. This pragma can only be used when the `VERSION2` or `VERSION3` C++ pragma is used. The C++ compiler

CPPCOMP generates an error if BUILD_NEUTRAL_LIBRARY is specified and the program references an object not marked as NEUTRAL in the corresponding VERSION2 or VERSION3 headers.

The BUILD_NEUTRAL_LIBRARY pragma can also be specified as a flag on the c89 or c99 command line as -Wbuild_neutral_library.

An object is neutral when the library and the application using it conform with these rules:

- The interface that the library provides uses only C linkage, or
- The interface that the library provides uses C++ linkage and all the parameters in these interfaces are marked as neutral or are strictly user-defined class types.
- Only these C++ standard library interfaces are shared directly or indirectly between the library and the user program:
    - global new operators
    - global delete operators
    - stdin, stdout, and stderr
    - errno
    - class std::exception
    - class std::bad_alloc

For example:

To create and link a library that uses the neutral C++ dialect and can be used by either a VERSION2 or VERSION3 program, enter:

```
CPPCOMP / IN ZCPPCDLL, OUT $S.#LIST / mydll; SHARED, &
   VERSION3, BUILD_NEUTRAL_LIBRARY
```

where ZCPPCDLL contains only interfaces common to both a VERSION2 and VERSION3 program; this is the low-level Standard C++ Library. The dynamic-link library mydll has its CPPNEUTRAL flag set for linker or loader checking.

Accessing Middleware Using HPE C and C++ for NonStop Systems

- Information about using NonStop SQL/MP and NonStop SQL/MX, which previously appeared in this section, is now contained only in these manuals:
    - *SQL/MP Programming Manual for C*
    - *SQL/MX Programming Manual for C and COBOL*

Using Tools.h++

The Tools.h++ class library is a C++ foundation class library, an industry standard available in the Guardian environment, in the OSS environment, and on the PC (Windows environment).

Two versions of Tools.h++ (version 7 and version 6.1) have been available since the D45 and G05 releases. Only version 7 is available on TNS/E systems. Migration information is available in Section 11, Collection Class Templates, of the *Tools.h++ User's Guide* for version 7.0. A summary is also available in **Migration Considerations for Version 7** on page 105.

Using Tools.h++ with the Standard C++ Library:

- you can use Tools.h++ version 6.1 only with `VERSION1` of the run-time library.
- you can use Tools.h++ version 7 only with `VERSION2` of the Standard C++ Library.
- you cannot use any version of Tools.h++ with the `VERSION3` Standard C++ Library, ISO/IEC 98, released at G06.20. However, much of the functionality of Tools.h++ is available in the `VERSION3` Standard C++ Library or can be easily built using components in `VERSION3`. For example, `RWCString` can be recoded using `string`. The Smalltalk collection is not directly supported in `VERSION3`, but can be coded by the user or ported from third-party software or open source.

User Documentation

- Version 6.1: *Tools.h++ Manual*
- Version 7: *Tools.h++ User's Guide and Tools.h++ Class Reference*

These manuals are available on the HPE NonStop user documentation disc using the HPE NonStop Technical Library (NTL) software.

Contents of the Tools.h++ Class Library (Version 7)

Tools.h++ version 7 is built on the Standard C++ Library, which is also available to HPE NonStop users. For more details, see **Using the Standard C++ Library** on page 88.

The Tools.h++ package for version 7 includes:

- Powerful single, multibyte, and wide character support
- Extended regular expressions
- Time and date handling classes

- Internationalization support
- Endian streams
- Multithread safe
- Persistent store
- Template-based classes
- Generic collection classes
- Smalltalk-like collection classes
- XDR streams for the OSS environment
- Other features of interest include:

    - RWFile class encapsulates standard file operations.
    - B-tree disk retrieval uses B-trees for efficient keyed access to disk records.
    - File Space Manager allocates, deallocates, and coalesces free space within a file.
    - A complete error handling facility takes advantage of C++ exceptions.
    - Additional classes include bit vectors, virtual I/O streams, cache managers, and virtual arrays.
    - Support has been added for IEEE floating-point arithmetic in version 7 of Tools.h++. Any routines that use `float` variables will automatically use the correct floating-point format, determined by the floating-point format you have specified. For more details, see pragma **IEEE_FLOAT** on page 248.

Migration Considerations for Version 7

Version 7 and version 6.1 are significantly different and are binarily incompatible. The collection class templates were reengineered between the two versions to make the templates compatible with the Standard C++ Library (`VERSION2`).

If you choose to move to version 7 and you have applications that were built using any earlier version of Tools.h++, you must recompile those programs.

To migrate existing code from version 6.1 to version 7, these code adjustments are required for specific classes:

- Extra template arguments are required for hashed and sorted collections. Use the predefined macro `RWDefHArgs` or `RWDefCArgs` to provide the additional template arguments.
- Different constructor arguments are required for hashed collections. You must use a hash object (an object that provides a hash function through a public member function) rather than a hashing function. Use the provided templatized object, `RWTHasher<T>`, which builds a hash object from a hash function.
- Element types in some collections may require less-than semantics (operator <()). This requirement is due to the inclusion of member functions based on the Standard C++ Library, such as `sort()`.
- Class hierarchies have changed. In version 7, there are no inheritance relationships among the standard-library-based collection class templates. If your code constructs an `RWTValHashTableIterator` from an `RWTValHashSet`, you should use an `RWTValHashSetIterator` instead.

Detailed migration information is available in Chapter 11 of the user guide portion of the *Tools.h++ Manual Version 7.*

Installation Notes

On an HPE NonStop system, the Tools.h++ library is delivered on the site update tape (SUT), and the files are installed on $SYSTEM. No user installation is required. On the PC, Tools.h++ is included on the CD for the Native C/C++ compiler.

Table 17: Installation Details for G06.20 Tools.h++

Environment	Location of Header Files	Location of SRLs or DLLS
Guardian	Version 6.1: $SYSTEM.ZRW	Version 6.1: $SYSTEM.SYSnn.ZTLHGSRL , $SYSTEM.SYSnn.ZTLHOSRL
	Version 7: $SYSTEM.ZINCRW70	G-series Version 7: $SYSTEM.SYSnn.ZTLHSRL
		H-series Version 7: $SYSTEM.ZDLL*nnn*.ZTLH7DLL
OSS	Version 6.1: `/usr/rogue6.1/rw`	In the Guardian namespace: Version 6.1: $SYSTEM.SYS*nn*.ZTLHGSRL , $SYSTEM.SYSnn.ZTLHOSRL
	Version 7: `/usr/rogue/rw`	G-series Version 7: $SYSTEM.SYSnn.ZTLHSRL
		H-series Version 7: $SYSTEM.ZDLL*nnn*.ZTLH7DLL

Including Header Files for Tools.h++

If your application uses any of the functions contained in the Tools.h++ library, you need to specify the header files for the library by adding `#include` directives either as preprocessor directives or as linker options.

For example, these `#include` directives specify three of the headers for the Version 7 `RWCstring` string class:

```
#include <rw/cstring.h>
#include <rw/regexp.h>
#include <rw/rstream.h>
```

For related MAPINCLUDE examples, see **Pragmas for Tools.h++** on page 108.

XDR Streams

Tools.h++ version 6.1 does not support the classes `RWXDRistream` and `RWXDRostream`.

Tools.h++ version 7 supports the classes `RWXDRistream` and `RWXDRostream` in both the OSS and Guardian environments.

Tools.h++ Example Files for Versions 6.1 and 7

Several example files are provided with the Tools.h++ header files. **Tools.h++ Example Files for Version 6.1** table lists the example files provided with version 6.1. The examples assume that the data files are in the current subvolume. BUS is an example of a SmallTalk Collection class.

Table 18: Tools.h++ Example Files for Version 6.1

Example File	Description of Contents
BUSC	BUS example source file
BUSH	BUS example header file
TEXTFILE	Example data file

For Version 7 of Tools.h++, the example files are located in the subphylum $system.zrw70 on the NonStop system. **Tools.h++ Example Files for Version 7** table lists the example files that are provided with Version 7.

Table 19: Tools.h++ Example Files for Version 7

Example File	Description of Contents
MAKEEXAM	MAKE file for the examples. It is an OBEY command file.
CPLUS	Macro used by MAKEEXAM to build each test. This must be modified to reflect the user's environment.
LINKIT	Contains instructions to `nld` for linking each test. This file is used by CPLUS. It must be modified to reflect the user's environment.
BINTREEC	Example source file
BTREEDSC	Example source file
BUSC	Example source file
BUSH	Example header file
CSTRINGC	Example source file
DEQUEVAC	Example source file
DOGH	Example header file
FMGRRTRC	Example source file
FMGRSAVC	Example source file

Table Continued

Example File	Description of Contents
GDLISTC	Example source file
HASHDICC	Example source file
I18NC	Example source file
MMAPPTRC	Example source file
PAGEHEAC	Example source file
TEXTFILE	Example data file
TIMEDATC	Example source file
TPDTESTC	Template example
TVDTESTC	Template example

Pragmas for Tools.h++

You need to use the `MAPINCLUDE` and `CPATHEQ` pragmas to map UNIX or OSS pathnames to Guardian file names. This examples illustrate the use of these pragmas:

1. The pragma:

   ```
   pragma MAPINCLUDE "from"="to"
   ```

   tells C or C++ to modify any #`include` name fully matching the "*from*" string. The effect is to replace the "*from*" string with the "*to*" string.

   - This example treats `#include` `"sys/types.h "` as `#include "$system.system.systypeh"`:

     ```
     PRAGMA MAPINCLUDE "sys/types.h"="$system.system.systypeh"
     ```

   - This example treats `#include <sys/types.h>` as `#include <systypeh>`:

     ```
     PRAGMA MAPINCLUDE "sys/types.h"="systypeh"
     ```

     This version of the pragma covers all cases where an exact match is required.

2. The pragma:

   ```
   pragma MAPINCLUDE PATH "from"="to"
   ```

   tells C or C++ to modify any #`include` name starting with the "*from*" string. It replaces the "*from*" string with the "*to*" string. Note that the slash is not a delimiter; it is part of the string.

   - This example treats `#include "rw/cstring.h"` as `#include "$system.rw.cstringh"`:

     ```
     PRAGMA MAPINCLUDE PATH "rw/"="$system.rw."
     ```

Note that `cstring.h` is not a Guardian file name. The compiler treats ".h" as "h".

- This example treats `#include <sys/type.h>` as `#include <systype.h>`:

  ```
  PRAGMA MAPINCLUDE PATH "sys/"="sys"
  ```

  This version of the pragma handles the problem of multiple versions of class libraries installed on a single system.

  Each set of class library include files from a given vendor uses the same directory for each class library provided. This is not a problem on small systems, because users typically install their own copy of the version they need for their own use. On large multiuser systems, however, it is more likely that users might want to use different versions of a specific class library.

  For example, to get version 7 of the Tools.h++ library, a user might specify:

  ```
  #pragma MAPINCLUDE PATH "rw/"= "$system.zincrw70."
  ```

  To get version 6.1 of the Tools.h++ library, a user might specify:

  ```
  #pragma MAPINCLUDE PATH "rw/"="$system.zrw."
  ```

3. The pragma:

`pragma MAPINCLUDE FILE "`*from*`"="`*to*`"`

tells C or C++ to modify any `#include` name ending with the "*from*" string. It replaces the "*from*" string with the "*to*" string.

This example treats `#include "strstream.h"` as `#include "strstreh"`:

```
PRAGMA MAPINCLUDE FILE "strstream.h"="strstreh"
```

This version of the pragma covers all cases where file names, other than Guardian file names, are longer than the longest allowable Guardian file name.

4. The pragma:

```
pragma CPATHEQ [ "filename" ]
```

defaults to searching for a file named "`cpatheq`" in the compiler's subvolume. You can supply your own `cpatheq` file as an optional argument. For version 7, HPE provides a CPATHEQ file named TLHMAP. However, no CPATHEQ file is provided for earlier versions of Tools.h++.

In this example, the `CPATHEQ` pragma tells the C++ compiler to open the $MYVOL.MYSUB.MYPATHEQ file and to process the information it contains:

```
PRAGMA CPATHEQ  "$myvol.mysub.mypatheq"
```

The contents of a CPATHEQ file should be in this suggested format:

```
#PRAGMA MAPINCLUDE      "sys/type.h"  = "systypeh"

#PRAGMA MAPINCLUDE PATH "rw/"         = "$system.zincrw70."

#PRAGMA MAPINCLUDE FILE "strstream.h" = "strstreh"
/*Standard C++ iostream header*/

#PRAGMA MAPINCLUDE FILE "iostream.h"  = "iostreah"
/*Standard C++ iostream header*/
```

```
#PRAGMA MAPINCLUDE FILE "rw/cacheman.h"  = "cachemah"
/*Rogue Wave Tools.h++ header*/
```

Usage Guidelines

- A `MAPINCLUDE FILE` pragma and a `MAPINCLUDE PATH` pragma can both operate on the same `#include` directive.
- Only one `MAPINCLUDE` pragma without the `PATH` or `FILE` modifier is performed on each `#INCLUDE` directive.
- `MAPINCLUDE FILE` and `PATH` pragmas can be specified more than once. Each one must have a unique "*from*" string, or a diagnostic message is issued.
- Subvolume searching by the SSV pragma does not change. That is, if an #include (after substitution) includes a volume or subvolume, it is not overridden by an SSV. The SSV is intended to expand file names that are not fully specified. For more details, see the description of pragma **SSV** on page 308.
- For version 7, there is a provided CPATHEQ file named TLHMAP. To use the TLHMAP file, enter this pragma:

  ```
  #pragma cpatheq "$system.zincrw70.tlhmap"
  ```

  For versions earlier than version 7, no CPATHEQ file is provided.
- The `CPATHEQ` and `MAPINCLUDE` pragmas are described further in the descriptions of the pragmas **CPATHEQ** on page 216 and **MAPINCLUDE** on page 266.

Boost Libraries Support

Boost is a collection of C++ class libraries. Most of these class libraries are implemented as templates and are self contained within a set of header files. A few Boost libraries require linking with a binary library. For more information on Boost libraries, see **http://www.boost.org/users/history/version_1_56_0.html**.

NonStop supports Boost version 1.56. NonStop support for Boost consists of providing Boost header files along with pre-built (as archive) binary files.

The NonStop port of the Boost headers supports the following:

- Compiling programs on OSS or Windows (using Windows hosted cross-compilers)
- TNS/E and TNS/X platforms
- OSS and Guardian environments
- C++ version 4
  - Version 4 is available only on TNS/X.
  - Version 3 is supported by both TNS/E and TNS/X.

Boost is included in the NonStop distribution. The installation will place the Boost distribution in `{$COMP_ROOT}/usr/boost_1_56` on NonStop systems and in `<compiler_install_directory>\usr\boost_1_56` on Windows systems.

A few libraries require a binary component. These are packaged as archives. Most Boost libraries are header-only templates that need not link with the Boost archives. The `boost_1_56/libv4` directory contains the archives compatible with C++ version 4. The archives are:

- `libboost`*`N`*`_prg_exec_monitor.a` –when building tests using the Boost Program Execution Monitor.
- `libboost`*`N`*`_test_exec_monitor.a` –when using the Boost Test Execution Monitor.
- `libboost`*`N`*`_unit_test_framework.a` –when using the Boost Unit Test Framework.
- `libboost`*`N`*`.a` – Contains the binary component for all other Boost libraries.

Where *N* represents the execution environment and can be `w`(TNS/X, LP64), `x`(TNS/X, ILP32), `y`(TNS/E, ILP64) or `z`(TNS/E, LP32).

Unsupported libraries or partially supported libraries

A few Boost libraries are closely aligned to specific operating systems, particularly Linux or Windows. These libraries are either unsupported or partially supported on NonStop.

The unsupported libraries are `asio`, `atomic`, `compatibility`, `context`, `coroutine`, `gil`, `interprocess`, `locale`, `log`, `MP1`, `python`, `signals`, `thread` and `TR1`.

The following libraries are partially supported:

- `container` –Extended Allocators are not supported.
- `iostreams` –Memory mapped files and `zlib` files are not supported.

Unsupported options

Boost is not supported with the following options:

- `—WTandem_float`
- `—Wversion1` and `—Wversion2`
- `-Wtarget=tns/r`

Compiling with Boost

When using Boost with the C++ compiler, add the `-Wboost` option to the compile command. The `-Wboost` option performs the following operations:

- Adds the Boost header file area to the list of directories that the compiler searches to locate `#include` files.
- Adds the appropriate `libboostN.a` archive to the set of objects supplied to the linker if `c89` or `c99` performs a link step. Consider the following while linking:

  - Programs which use the test library and require a test library archive must explicitly include the archive in link step. For example, `${COMP_ROOT}/usr/boost_1_56/libv3/libboostN_test_exec_monitor.a`.
  - Programs compiled with c89 or c99 utility use the boost libraries in the boost_1_56/libv3 area. Programs compiled with c11 utility use the boost libraries in the boost_1_56/libv4 area.
  - The Boost archives require the routines defined in the C++ standard libraries. If `.c`, `.cpp`, `.cc`, or `.cxx` files are not specified in the `c89` or `c99`, use the `–Wcplusplus` option to ensure the C++ standard libraries are included in the link step.

- Shortens mangled names that depend upon entities defined in the boost namespace to 2048 characters. CRC encoding is used to ensure that the shortened names are unique. However, the shortened names cannot be de-mangled. Boost mangled names tend to be excessively large, many times exceeding 10000 characters. Shortening the names uses fewer resources (less memory) during compilation and uses less space in the resulting object file.

NOTE: A few open source software searches for Boost headers in the standard `/usr/include/boost` area. NonStop places Boost headers in `/usr/boost_1_56/boost` folder. Modify the open source procedures to search for Boost headers in the NonStop location. Modify makefiles to add the `–Wboost` option to the compile commands.

Mixed-Language Programming for TNS Programs

This chapter describes the Common Run-Time Environment (CRE) and the C interface declarations that are necessary to interface to other programming languages. The CRE and C interface declarations are HPE features for NonStop systems that are not available in standard ISO/ANSI C.

TNS programs in the Guardian environment can contain routines written in TNS COBOL, FORTRAN, TAL, TNS C, and D-series Pascal. TNS programs in the OSS environment can contain routines written in TNS COBOL, TAL, TNS C, and TNS C++.

Applications that include modules written in other languages can also be compiled in the Guardian environment to run in the OSS file system (see **Binding a C Module** on page 340). For a comparison of TNS and TNS/R native mixed-language programming, see **Differences Between Native and TNS Mixed-Language Programs** on page 161.

Introducing the CRE

The Common Run-Time Environment (CRE) is a set of services that supports mixed-language programs. The CRE library is a collection of routines that implements the CRE. The CRE library enables the language-specific run-time libraries to coexist peacefully with each other. User routines and run-time libraries call CRE library routines to access shared resources managed by the CRE, such as the standard files (input, output, and log) and the user heap, regardless of language.

The CRE does not support all possible operations. For example, the CRE supports file sharing only for the three standard files: standard input, standard output, and standard log. The language-specific run-time libraries access all other files by calling Guardian system procedures directly, whether or not a program uses the CRE.

C-series programs run in a language-specific run-time environment. D-series C and Pascal programs run only in the CRE. D-series COBOL, FORTRAN, and TAL programs can run in either their language-specific run-time environments or the CRE. All the routines in a program must be compiled to run in either a language-specific run-time environment or the CRE.

Run-Time Environments Available table lists the C-series and D-series run-time environments available for each language.

Table 20: Run-Time Environments Available

Language	C-Series Environment	D-Series Environments
C	C run-time environment	CRE
COBOL	COBOL run-time environment	CRE or COBOL run-time environment
FORTRAN	FORTRAN run-time environment	CRE or FORTRAN run-time environment
D-series Pascal	Pascal run-time environment	CRE
TAL	No run-time environment	CRE or no run-time environment

The language of a program's main routine determines the program's run-time environment. (The main routine is the routine that executes first in your program; it is the routine declared with the `main` keyword, or in some languages the PROGRAM keyword.)

- If a program does not use the CRE, routines written in a language other than that of the main routine have limited access to their run-time libraries. For example, if:
  - The main routine of a mixed-language program is written in COBOL
  - The program runs in a language-specific run-time environment (the COBOL run-time environment)

  then a C routine has limited access to the TNS C run-time library.
- If a program uses the CRE, each routine in the program appears to be running in its own language-specific run-time environment, regardless of the language of the main routine. For example, if:
  - The main routine of a mixed-language program is written in COBOL
  - The program runs in the CRE

  then a C routine has complete access to the TNS C run-time library.

D-series TNS C and Pascal run-time libraries always call CRE library routines for services managed by the CRE; they can run only in the CRE. D-series TNS COBOL and FORTRAN run-time libraries call CRE library routines if you compile all the routines in a program to run in the CRE. In contrast, TAL routines must call CRE library routines directly.

For information on writing programs that use the services provided by the CRE, see the *Common Run-Time Environment (CRE) Programmer's Guide*. For more details on specifying a run-time environment, see the description of pragma **ENV** on page 221. For more details on mixed-language binding, see **Restrictions on Binding Caused by the ENV Pragma** on page 343.

Using Standard Files in Mixed-Language Programs

In a mixed-language program, if a TNS C function is to be the main function, it should be compiled with the `NOSTDFILES` pragma to keep it from automatically opening the three standard C files: `stdin`, `stdout`, and `stderr`.

If the main routine is not written in C, the three standard C files will not be automatically opened. If you want any or all the standard files to be opened for C, you must explicitly open them by calling the `fopen_std_file()` function.

Writing Interface Declarations

Your TNS C programs can call procedures written in C++, COBOL, FORTRAN, D-series Pascal, and TAL, in addition to procedures written in an unspecified language type. You cannot mix TNS and native-mode language modules.

All external procedures must be declared. The interface declaration indicates the correct language or indicates "unspecified," if the language is unknown. If the language is unspecified, the external procedure is assumed to be written in C.

Because lexical and operational features differ between C and these other languages, you must use an interface declaration instead of a simple function declaration to declare a procedure written in one of these other languages. Interface declarations are an HPE extension to the ISO/ANSI C standard for function declarations. They provide additions that account for most of the differences between C and the other language.

The modern, preferred method for writing an interface declaration is to use a standard function prototype followed later by a FUNCTION pragma. This is the general form:

```
int NAME (<params>);
...
#pragma function NAME (params)
```

For more details on syntax, see the pragma **FUNCTION** on page 239. For an illustration of a FUNCTION pragma used in an interface declaration, see the examples following this discussion.

The older method for declaring procedures is to declare the procedure just as you would a C function, except that you include:

- A language attribute specifier (`_c, _cobol, _fortran, _pascal, _tal,` or `_unspecified`) to identify the language of the external procedure.
- An `_alias` attribute specifier to assign the external name, or rather the name as known to other language.

The syntax for these older-style interface declarations is described under **Attribute Specifier** on page 61. You should use the `FUNCTION` pragma to declare external routines.

After providing an interface declaration, your TNS C program uses normal function calls to access the procedure written in the other language. However, remember that these calls cross language boundaries; therefore, there are restrictions beyond those of normal C function calls.

The C interface declaration enables you to declare most types of COBOL, FORTRAN, Pascal, and TAL procedures, including:

- D-series Pascal and TAL procedures defined with the VARIABLE or EXTENSIBLE attribute
- Procedures whose names are not valid C identifiers
- TAL procedures that do not return a value but return a condition code
- COBOL, FORTRAN, D-series Pascal, and TAL procedures with extended pointer (.EXT) parameters

However, your TNS C program cannot directly call a TAL procedure that both returns a value and returns a condition code. For a description of techniques that enable your TNS C program to access TAL procedures that fall into this category, see **TAL Procedures That You Cannot Directly Call** on page 123.

Usage Guidelines

- Procedure names

  When you specify the C name of a non-C procedure, you should use the non-C name of the procedure if it is a valid C identifier. If the name is not a valid C identifier because it includes circumflexes (^) or hyphens (-), simply substitute underscores (_) for the circumflexes and hyphens in the C name.

- Return value types

  For procedures that return a scalar value, you should declare the C counterpart to the TAL scalar type as the procedure type in the interface declaration. **Compatible TAL and TNS C DATA Types** shows the C counterparts to TAL scalar types.

For procedures that return a condition code, you should declare _cc_status as the procedure's return type in the interface declaration. The tal.h header file contains three macros to interrogate the results of the procedure declared with the _cc_status type specifier:

```
#define _status_lt(x) ((x) <  0)
#define _status_le(x) ((x) <= 0)
#define _status_eq(x) ((x) == 0)
#define _status_ge(x) ((x) >= 0)
#define _status_gt(x) ((x) >  0)
#define _status_ne(x) ((x) != 0)
```

Before you can use these macros, you must include the tal.h header file.

Note that you should avoid designing TAL procedures that return a condition code, because that is an outdated programming practice. Guardian system procedures must retain this interface for reasons of compatibility.

- Parameter types

  When creating an interface declaration for a non-C procedure, you must ensure that the declared C type of a parameter matches the defined type of that parameter. The TNS C compiler cannot perform this task automatically, because it does not have direct access to each language's definition. The TNS C compiler does, however, check that the type of an argument in a call to a non-C procedure matches its corresponding parameter's type. Ensuring that the C and non-C types of a scalar parameter match is a simple task, because most scalar types have direct counterparts in C, as shown in **Data Type Correspondence** on page 559.

- When you invoke a TNS COBOL procedure, COBOL always returns `void` because the COBOL language has no way to declare a COBOL program to be a function. Therefore, you must always specify TNS COBOL routines with type `void`.
- TNS COBOL programs use 32-bit addressing for all data items in the Extended-Storage Section and all data items in the Linkage Section that are not described as having ACCESS MODE of STANDARD.
- If the TAL or D-series Pascal definition specifies the EXTENSIBLE attribute, the interface declaration must specify _ `extensible`.
- If the TAL or D-series Pascal definition specifies the VARIABLE attribute, the interface declaration must specify _ `variable`.

Examples

1. This example, taken from the `stdlib.h` header file, shows the preferred style for an interface declaration (in this case, for the CRE_ASSIGN_MAXORDINAL_ procedure). The interface declaration consists of a standard function prototype and a `FUNCTION` pragma:

```
void get_max_assign_msg_ordinal (void);
...
#pragma function get_max_assign_msg_ordinal \
(alias("CRE_ASSIGN_MAXORDINAL_"), extensible, tal)
```

The example is equivalent to this older-style interface declaration:

```
_extensible _tal _alias (“CRE_ASSIGN_MAXORDINAL_”) \
get_max_assign_msg_ordinal (void);
```

2. This example, taken from the `cextdecs` header file, shows the interface declaration for the CHANGELIST system procedure. Notice that identifier names in the parameter type list are optional because C only cares about the types of the parameters:

```
_tal _variable _cc_status CHANGELIST(short, short, short);
```

3. In this example, the parameter type must be followed by the _far specifier because the TAL parameter was declared as an extended pointer using .EXT:

```
_tal _variable _cc_status PROC_3 (short _far *);
```

4. This example shows you how to use the _cc_status type specifier. Notice that the variable `c_code` is declared as type `short` so that it can be compared against the condition code that is contained in the _status_eq() macro:

```
_tal _cc_status _alias ("C^GETPOOL") C_GETPOOL
                                    (short _far *pool_head,
                                    long block_size,
                                    long *block_addr);

short *pool_alloc(long size)

{
  short c_code;
  long blk_addr;

  c_code = C_GETPOOL(my_pool,size,&blk_addr);
  if (__status_eq(c_code))
    return( (short *)blk_addr );
  else
    return( NULL );
}
```

5. This function prototype and FUNCTION pragma:

```
void segment_allocate (short, long, short *, short);
...
#pragma function segment_allocate (tal, alias
(“SEGMENT_ALLOCATE_”), variable)
```

is equivalent to this interface declaration:

```
_tal _variable short SEGMENT_ALLOCATE_ (short, long,
                                       short *, short);
```

6. Here are some more examples of interface declarations for calling TAL procedures:

```
_tal _variable _cc_status SEGMENT_DEALLOCATE_ (short, short);

_tal _variable _cc_status READ (short, short *, short,
                               short *, long);

_tal _extensible _cc_status READX (short, char _far *,
                                  short, short *, long);
```

```
_tal void c_name = "tal^name" (short *);
```

Interfacing to TAL

This subsection provides guidelines for writing programs composed of TAL and HPE TNS C modules for NonStop systems. This discussion assumes that you have a working knowledge of TAL and C and are familiar with the contents of these manuals:

- *TAL Programmer's Guide*
- *TAL Reference Manual*

For information on calling TAL routines from another language, see the manual for TNS COBOL, FORTRAN, or D-series Pascal.

Using Identifiers

TAL and C identifiers differ:

- TAL and C have independent sets of reserved keywords.
- TAL identifiers can include circumflexes (^), whereas C identifiers cannot.
- The C compiler is case-sensitive; the TAL compiler is not case-sensitive.

To declare variable identifiers that satisfy both compilers:

- Avoid using reserved keywords in either language as identifiers.
- Specify TAL identifiers without circumflexes.
- Specify C identifiers in uppercase.

You can declare TAL-only or C-only routine identifiers and satisfy both compilers by using the public name option in:

- Interface declarations in C
- EXTERNAL procedure declarations in TAL

In Inspect sessions:

- Use uppercase for TAL identifiers
- Use the given case for C identifiers

In Binder sessions, use mode noupshift for lowercase C identifiers.

Matching Data Types

Use data types that are compatible between languages for:

- Shared global variables
- Formal or actual parameters
- Function return values

Compatible TAL and TNS C Data Types table lists compatible TAL and C data types for each TAL addressing mode.

Table 21: Compatible TAL and TNS C Data Types

TAL Addressing Mode	TAL Data Type	C Data Type	Notes
Direct	STRING	`char`	
Direct	INT	`short`	TAL INT signed range only
Direct	INT(32)	`long`	
Direct	FIXED(0)	`long long`	TAL FIXED(0) only
Direct	REAL	`float`	
Direct	REAL(64)	`double`	
Standard indirect (.)	STRING	`char *`	
Standard indirect (.)	INT	`short *`	
Standard indirect (.)	INT(32)	`long *`	
Standard indirect (.)	FIXED(0)	`long long *`	
Standard indirect (.)	REAL	`float *`	
Standard indirect (.)	REAL(64)	`double *`	
Extended indirect (.EXT)	STRING	`char _far *`	For `_far`, see Note.
Extended indirect (.EXT)	INT	`short _far *`	
Extended indirect (.EXT)	INT(32)	`long _far *`	
Extended indirect (.EXT)	FIXED(0)	`long long _far *`	
Extended indirect (.EXT)	REAL	`float _far *`	
Extended indirect (.EXT)	REAL(64)	`double _far *`	

NOTE: In C, use _far only in the parameter-type list of an interface declaration to specify a parameter type that is defined in TAL as an extended pointer.

Incompatibilities between TAL and TNS C data types include:

- TAL has no numeric data type that is compatible with C unsigned `long` type.
- TAL UNSIGNED is not compatible with C `unsigned short` type. TAL UNSIGNED(16) can represent signed or unsigned values.

For more details on TNS C and TAL data types, see **Variables and Parameters** on page 127.

Memory Models

A TNS C program can use the small-memory model or the large-memory model, depending on the amount of data storage required. The large-memory model is recommended and is the default setting. All examples in this subsection illustrate the large-memory model unless otherwise noted.

A TAL program can use any of these memory combinations, depending on the application's needs:

- The user data segment
- The user data segment and the automatic extended data segment
- The user data segment and one or more explicit extended data segments
- The user data segment, the automatic extended data segment, and one or more explicit extended data segments

This table describes some aspects of memory usage by C and TAL programs. The far right column refers to the upper 32K-word area of the user data segment.

Language	Memory Model	Addressing	Data Storage	Upper 32K‑Word Area
TNS C	Small	16-bit	32K words	Reserved
TNS C	Large	32-bit	127.5 MB	Reserved
TAL	N.A.	16-bit or 32-bit	64K words (without the CRE), plus 127.5 MB in each extended data segment that is allocated	Reserved only if you use the CRE

Any TAL module that uses the upper 32K-word area of the user data segment cannot run within a TNS C object file that contains the main routine.

Data Model

The size of the C data type `int` is 16 bits in the 16‑bit data model and 32 bits in the 32‑bit data model. (The 32‑bit data model is also called the wide-data model.) If you specify the 32‑bit data model, the C data type `int` is 32 bits and the TAL data type INT corresponds to the C data type `short`.

Any interface to a TAL routine must be specified in such a way that there is no possibility for a data-length mismatch. Therefore, in your C program, use `short` for a 16-bit integer and `long` for a 32-bit integer and avoid the use of `int`. Using `short` in your C program enables you to use the same declarations regardless of the data model your program uses. The type `short` is always a 16-bit integer in the C compiler.

Calling TNS C Routines From TAL Modules

A TAL module must include an EXTERNAL procedure declaration for each TNS C routine to be called. This TAL code shows the EXTERNAL procedure declaration for C_FUNC and a routine that calls C_FUNC. ARRAY is declared with .EXT, because C_FUNC uses the large-memory model:

```
TAL Code                             C Code


INT status := 0;             short C_FUNC(char *str)
STRING .EXT array[0:4];      {
                                *str = 'A';
INT PROC c_func (a)             str[2] = 'C';
  LANGUAGE C;                   return 1;
    STRING .EXT a;           }
EXTERNAL;


PROC example MAIN;
  BEGIN
  array[2] := "B";
  status := c_func (array);
  array[1] := "B";
  END;
```

A C-series TNS C module called by a TAL module has limited access to the C run-time library. If the TNS C module needs full access to the C run-time library, you can either:

- Modify the program to run in the CRE as described in this section.
- Specify a C MAIN routine that calls the original TAL main routine.

  In the TAL module, remove the MAIN keyword from the TAL main routine and remove any calls to the INITIALIZER or ARMTRAP system procedure. The TAL module must also meet the requirements of the C run-time environment.

```
TAL Code                             C Code

                         #include <stdioh> nolist
INT status := 0;
INT .EXT array[0:4];     _tal void TALMAIN ( void );

INT PROC cfunc (a)       short CFUNC (short *num)
  LANGUAGE C;            {
    INT .EXT a;             printf("num B4=%d\n",*num);
EXTERNAL;                   num[0] = 10;
                            printf("num AF=%d\n",*num);
PROC talmain;               return 1;
  BEGIN                  }
  array[2] := 2;
  status :=              main ()   /* C MAIN routine */
      cfunc (array);     {
  END;                     TALMAIN ();
                         }
```

Calling TAL Routines From TNS C Modules

A TNS C module has full access to the TNS C run-time library even if the C module does not contain the `main()` routine.

When you code TNS C modules that call TAL routines:

- Include an interface declaration for each TAL routine to be called, or a function prototype and a FUNCTION pragma, as described in **Using a Function Prototype and a FUNCTION Pragma** on page 142.
- If a called TAL routine sets the condition code, include the `talh` header file.
- If a called routine is a system procedure, include the `cextdecs` header file.

In C, interface declarations are comparable to EXTERNAL procedure declarations in TAL.

To specify an interface declaration in C, include:

- The keyword `_tal`
- The `_variable` or `_extensible` attribute, if any, of the TAL routine
- The data type of the return value, if any, of the TAL routine
- A routine identifier
- A public name if the TAL identifier is not a valid C identifier
- A parameter-type list or, if no parameters, the keyword void
- For extended pointers in the parameter-type list, the keyword `_far` after the parameter type

The return type value can be any of these:

Return Type Value	**Meaning**
void	The TAL routine does not return a value.
fundamental-type	The TAL routine returns a value. Specify one of, or a pointer to one of, the character, integer, or floating-point types.
`_cc_status`	The TAL routine does not return a value but does set a condition code. The `tal.h` header file contains six macros that interrogate this condition code: `_status_lt(x)`, `_status_le(x)`, `_status_eq(x)`, `_status_ge(x)`, `_status_gt(x)`, and `_status_ne(x)`.

For information on calling TAL routines that both return a value and set a condition code (CC), see **TAL Procedures That You Cannot Directly Call** on page 123.

Here are examples of interface declarations for calling TAL routines:

```
_tal _variable short SEGMENT_ALLOCATE_ (short, long,
                                       short *, short);
_tal _variable _cc_status SEGMENT_DEALLOCATE_ (short, short);
_tal _variable _cc_status READ (short, short *, short,
                             short *, long);
_tal _extensible _cc_status READX (short, char _far *,
                                 short, short *, long);
_tal _alias ("tal^name") void c_name (short *);
```

After specifying an interface declaration, use the normal C routine call to access the TAL routine.

This example shows a large-memory-model TNS C module that calls a TAL routine:

C Code

```
#include <stdioh> nolist
short arr[5];        /*stored in extended segment */
_tal _alias ("tal^name") void C_Name (short _far *);
void func1 (short *xarr)
{
  C_Name (xarr);
  printf ("xarr[2] after TAL = %d", xarr[2]);
}
main ()
{
  arr[4] = 8;
  func1 (arr);
}
```

TAL Code

```
PROC tal^name (a);
    INT .EXT a;          !32-bit pointer
  BEGIN
  a[2] := 10;
  END;
```

TAL Procedures That You Cannot Directly Call

Your TNS C program cannot directly call a TAL procedure that both returns a value and sets the condition code register.

To access a TAL procedure that both returns a value and sets a condition code, you must write a “jacket” procedure in TAL that is directly callable by your TNS C program. You define this jacket procedure so that it:

- Passes arguments from C calls through to the TAL procedure unchanged
- Returns the TAL procedures return value indirectly through a pointer parameter
- Returns the condition code using the function-return type `_cc_status`

To illustrate this technique, this example defines a jacket procedure for the GETPOOL system procedure. First, here is the TAL source module that defines the jacket routine:

```
?SOURCE EXTDECS(GETPOOL);

PROC C^GETPOOL(pool^head, block^size, block^addr);
  INT .EXT pool^head;    ! Address of pool head
  INT(32) block^size;    ! Block size
  INT(32) .block^addr;   ! Block address

BEGIN
  ! Call GETPOOL, storing return value in block^addr
  block^addr := GETPOOL(pool^head, block^size);
  RETURN;  ! No return value here
END;
```

The definition of C^GETPOOL declares the same parameters as the GETPOOL system procedure, with the addition of block^addr. This additional parameter is a pointer to the return type of GETPOOL, which is

INT(32). Note that the RETURN statement specifies no return value and that it immediately follows the GETPOOL call, therefore ensuring that the condition code is unchanged.

Here is the interface declaration for C^GETPOOL as you would enter it in your C module:

```
_tal _cc_status _alias ("C^GETPOOL") C_GETPOOL
                                    (short _far *pool_head,
                                    long block_size,
                                    long *block_addr);
```

Once you have made this declaration, you can effectively call GETPOOL by calling the jacket routine C_GETPOOL; for example:

```
short *pool_alloc(long size)

{
  short c_code;
  long blk_addr;

  c_code = C_GETPOOL(my_pool,size,&blk_addr);
  if (_status_eq(c_code))
    return( (short *)blk_addr );
  else
    return( NULL );
}
```

Sharing Data

You can share global data in the user data segment between these types of TAL and TNS C modules:

- TAL modules that declare global variables having standard indirection (.)
- TNS C small-memory-model modules

You can share global data in the automatic extended data segment between these types of TAL and C modules:

- TAL modules that declare global variables having extended indirection (.EXT)
- TNS C large-memory-model modules

In a large-memory-model TNS C module, you can use the `_lowmem` storage class specifier to allocate a C array or structure that can be represented by a 16-bit address if needed in a call to a TAL routine or a system procedure.

Using pointers to share data is easier and safer than trying to match declarations in both languages. Using pointers also eliminates problems associated with where the data is placed.

To share data by using pointers, first decide whether the TAL module or the TNS C module is to declare the data:

- If the TAL module is to declare the data, follow the guidelines in **Sharing TAL Data With TNS C Using Pointers** on page 124.
- If the TNS C module is to declare the data, follow the guidelines in **Sharing TNS C Data With TAL Using Pointers** on page 125.

Sharing TAL Data With TNS C Using Pointers

To share TAL global data with TNS C modules, follow these steps:

1. In the TAL module, declare the data using C-compatible identifiers, data types, and alignments. (Alignments depend on byte or word addressing and variable layouts as described in **Variables and Parameters** on page 127.)

   When you declare TAL arrays and structures, use indirect addressing.

2. In the TNS C module, declare pointers to the data, using TAL-compatible data types.
3. In the TNS C module, declare a routine to which TAL can pass the addresses of the shared data.
4. In the C routine, initialize the pointers with the addresses sent by the TAL module.
5. Use the pointers in the TNS C module to access the TAL data.

This example shows how to share TAL data with a large-memory-model TNS C module. The TAL module passes to a C routine the addresses of two TAL arrays. The C routine assigns the array addresses to C pointers.

C Code

```
short *c_int_ptr;            /* pointer to TAL data */
char *c_char_ptr;            /* pointer to TAL data */
short INIT_C_PTRS (short *tal_intptr, char *tal_strptr)
{                            /* called from TAL */
  c_int_ptr = tal_intptr;
  c_char_ptr = tal_strptr;
  return 1;
}
/* Access the TAL arrays by using the pointers */
```

TAL Code

```
STRUCT rec (*);
  BEGIN
  INT x;
  STRING tal_str_array[0:9];
  END;
INT .EXT tal_int_array[0:4];     !TAL data to share with C
STRUCT .EXT tal_struct (rec);    !TAL data to share with C
INT status := -1;
INT PROC init_c_ptrs (tal_intptr, tal_strptr) LANGUAGE C;
    INT .EXT tal_intptr;
    STRING .EXT tal_strptr;
  EXTERNAL;
PROC tal_main MAIN;
  BEGIN
  status := init_c_ptrs
                 (tal_int_array, tal_struct.tal_str_array);
  !Do lots of work
  END;
```

Sharing TNS C Data With TAL Using Pointers

To share TNS C global data with TAL modules, follow these steps:

1. In the TNS C module, declare the data using TAL-compatible identifiers, data types, and alignments. (Alignments depend on byte or word addressing and variable layouts as described in **Variables and Parameters** on page 127.)

   C arrays and structures are automatically indirect.

2. In the TAL module, declare pointers to the data, using C-compatible data types.
3. In the TAL module, declare a routine to which C can pass the addresses of the shared data.
4. In the TAL routine, initialize the pointers with the addresses sent by C.
5. Use the pointers in the TAL module to access the C data.

This example shows how to share C data with a TAL module. The TNS C module passes the addresses of two C arrays to a TAL routine. The TAL routine assigns the array addresses to TAL pointers.

C Code

```
#include <stdioh> nolist

short arr[5];                   /* C data to share with TAL */
char charr[5];                  /* C data to share with TAL */
_tal void INIT_TAL_PTRS ( short _far *, char _far * )
_tal _alias ("tal^name") void C_Name (void);
void example_func( int *x )
{
  printf("x before TAL = %d\n", x[2] );
  C_Name( );
  printf("x after TAL = %d\n", x[2] );
}
main ()
{
  INIT_TAL_PTRS ( &arr[0], &charr[0] ); /* initialize ptrs */
    /* test pointer values */
  arr[0] = 8;
  example_func( arr );
  arr[2] = 18;
  charr[2] = 'B';
}
```

TAL Code

```
INT .EXT tal_int_ptr;                     !Pointer to C data
STRING .EXT tal_char_ptr;                 !Pointer to C data

PROC init_tal_ptrs (c_addr1, c_addr2);     !Called from C
    INT .EXT c_addr1;
    STRING .EXT c_addr2;
  BEGIN
  @tal_int_ptr := @c_addr1;
  @tal_char_ptr := @c_addr2;
  END;


PROC tal^name;
  BEGIN
  tal_int_ptr[0] := 10;
  tal_int_ptr[2] := 20;
  tal_char_ptr[2] := "A";
  END;
```

Sharing TAL Data With TNS C Using BLOCK Declarations

As of the D20 release, TAL modules can share global data with TNS C modules by declaring each shared variable in its own BLOCK declaration and giving both the variable and the BLOCK the same name. The TNS C modules must also declare each shared variable; the layout of the variable must match in both the TAL and C modules.

In this example, a TAL module declares a variable within a BLOCK declaration, and the TNS C module declares the equivalent variable:

```
TAL Code                             C Code

NAME TAL_module;
BLOCK fred;
  INT .EXT fred;              int FRED;  /*all uppercase*/
  END BLOCK;
```

Because this method requires that the layout of the corresponding TAL and C declarations match, it is recommended that you share data by using pointers where possible.

Variables and Parameters

This subsection gives guidelines for declaring compatible TAL and C variables and parameters. These guidelines supplement those given in **Sharing TAL Data With TNS C Using BLOCK Declarations** on page 127. The topics are discussed are:

- STRING and `char` variables
- Arrays
- Structures
- Multidimensional arrays
- Arrays of structures
- Redefinitions and unions
- Pointers
- Enumeration variables
- Bit-field manipulation
- UNSIGNED variables and packed bit fields
- TAL routines as parameters to C
- C routines as parameters to TAL

When you declare formal reference parameters, remember to use indirection:

- If the caller is a small-memory-model TNS C routine, use standard indirection (.) for the TAL formal parameter.
- If the caller is a large-memory-model TNS C routine, use extended indirection (.EXT) for the TAL formal parameter.

STRING and char Variables

TAL STRING and C `char` simple variables each occupy one byte of a word. These are STRING and `char` compatibility guidelines:

- Share variables of type TAL STRING and C char by using pointers.
- Declare TAL STRING and C char formal parameters as reference parameters to avoid these value-parameter incompatibility:
    - When you pass a STRING parameter to a C routine, the actual byte value occupies the left byte of the word allocated for the C `char` formal parameter.
    - When you pass a `char` parameter to a TAL routine, the actual byte value occupies the right byte of the word allocated for the TAL STRING formal parameter.

For example, if you declare a TAL STRING formal parameter as a value parameter rather than as a reference parameter, the TAL routine can access the C `char` actual parameter only by explicitly referring to the right byte of the word allocated for the STRING formal parameter:

```
PROC sample (s);
    STRING s;              !Declare TAL STRING parameter as a
  BEGIN                    ! value (not reference) parameter
  STRING dest;
  dest := s[1];            !Refer to right byte of word
  END:
```

Arrays

TAL and C arrays differ:

Characteristic	TAL Array	C Array
Lower bound	Any integer	Always zero
Dimensions	One dimension *	One or more dimensions
Direct or indirect	Direct or indirect	Indirect only
Byte or word addressing	STRING arrays and extended indirect arrays are byte addressed; all other arrays are word addressed	`char` arrays and large-memory-model arrays are byte addressed; all other arrays are word addressed

* TAL structures can emulate multidimensional C arrays, as discussed in **Multidimensional Arrays** on page 129.

To declare compatible TAL and C arrays:

- Use data types and alignments that satisfy both compilers.
- Declare TAL arrays that have a lower bound of 0.
- Declare one-dimensional C arrays.
- Declare indirect TAL arrays.

These are compatible arrays in TAL and TNS C (large-memory model):

```
TAL Code                       C Code
INT .EXT robin[0:9];        short robin [10];
INT(32) .EXT gull[0:14];    long gull [15];
STRING .EXT grebe[0:9];      char grebe [10];
```

Structures

All TAL and C structures begin on a word boundary. These are guidelines for sharing TAL and C structures and passing them as parameters:

- Specify the same layout for corresponding TAL and C structures.
- Specify compatible data types for each item of both structures.
- In TAL, pass structures by reference.
- In C, use the & (ampersand) operator.
- In TAL, a routine cannot return a structure as a return value.

This TAL and C structures have compatible layouts:

```
TAL Code                        C Code
STRUCT rec (*);              struct birdname
  BEGIN                       {
  INT x;                        short x;
  STRING y[0:2];                char y[3];
  END;                        } robin[10];
STRUCT .EXT robin(rec)[0:9];
```

This TAL and C structures have compatible layouts:

```
TAL Code                      C Code
STRUCT rec1 (*);           struct rec1
  BEGIN                     {
  STRING a, b, c;            char a, b, c;
  END;                     };
```

This TAL and C structures also have compatible layouts:

```
TAL Code                      C Code
STRUCT rec2 (*);             struct rec2
  BEGIN                        {
  STRING e;                     char e;
  INT y;                        short y;
  STRING g;                     char g;
  END;                         };
```

Multidimensional Arrays

In C, you can declare multidimensional arrays. In TAL, you can emulate multidimensional arrays by declaring structures that contain arrays.

Here is an example of multidimensional arrays in TAL and TNS C (large-memory model):

```
TAL Code                       C Code
STRUCT rec1 (*);
  BEGIN
  INT y[0:4];                short cma[10][5];
  END;
```

```
STRUCT .EXT tma(rec1)[0:9];
!Sample access!               /* sample access */
tma[8].y[3] := 100;            cma[8][3] = 100;
```

Arrays of Structures

If you specify bounds when you declare a TAL structure, you create an array of structures. This TAL and C arrays of structures are equivalent. Each declaration contains an array of ten structure occurrences:

```
TAL Code                        C Code
STRUCT cell (*);             struct cell
  BEGIN                      {
  INT x;                       short x;
  STRING y;                    char y;
  END;                        };

STRUCT .EXT tcell(cell)[0:9]; struct cell ccell [10];
PROC honey (c);                void JOANIE
    INT .EXT c (cell);         (struct cell *);
EXTERNAL;
```

Redefinitions and Unions

Variant records are approximated by TAL structure redefinitions and C unions. A TAL redefinition declares a structure item that uses the same memory location as an existing structure item. The existing structure item can be a simple variable, array, substructure, or pointer that:

- Begins on a word boundary
- Is at the same BEGIN-END level in the structure as the redefinition
- Is the same size or larger than the redefinition

A C union defines a set of variables that can have different data types and whose values alternatively share the same portion of memory. The size of a union is the size of its largest variable; the largest item need not come first. A union always begins on a word boundary.

Pointers

Pointers contain memory addresses of data. You must store an address into a pointer before you use it. In TAL and C pointer declarations, specify the data type of the data to which the pointer points. You must use pointers when sharing global variables. You can pass pointer contents by value between TAL and C routines.

Differences between TAL and C pointers include:

- TAL structure pointers can point to a byte or word address.
- C structure pointers always point to a word address. To pass a C structure pointer to a TAL routine that expects a byte structure pointer, you must explicitly cast the C pointer to type `char`.
- TAL pointers are dereferenced implicitly.
- C pointers are usually dereferenced explicitly.
- Small-memory-model C routines use 16-bit pointers only.

- Large-memory-model C routines use 32-bit pointers only, even if the pointers refer to the user data segment. In global structure declarations, you must specify `_lowmem` in the storage class of the declaration.
- If a TAL routine expects a 16-bit pointer, the C pointer you pass must refer to an object in user data space.

Here are examples of TAL and TNS C pointers (large-memory model):

```
TAL Code                          C Code



STRUCT rec (*);                 struct rec
  BEGIN                         {
  INT d;                        short d;
  INT .p (rec);                 struct rec *p;
  END;                            };
BLOCK joe;
  INT .EXT joes (rec);          struct rec *JOE;
  END BLOCK;
PROC tonga (p);                 void CALEDONIA
    INT .EXT p (rec);           (struct rec *p)
  BEGIN                         {
  !Lots of code                 /* Lots of code */
  END;                            }
```

Each language can call the other, passing the address in the pointer by value:

```
TAL Code                                          C Code
CALL caledonia (joes);           TONGA (joe);
```

Here are examples of TAL and TNS C structure pointers (large-memory model) that implement a linked list:

```
TAL Code                           C Code

STRUCT rec (*);                  struct rec
  BEGIN                          {
  INT x;                            short x;
  INT .EXT strptr (rec);            struct rec *p;
END;                              };
STRUCT .EXT joe (rec);            struct rec joe;
PROC callme (param1);            void f1 (struct rec *);
    INT .EXT param1 (rec);
EXTERNAL;
```

Enumeration Variables

Using C enumeration variables, you can associate a group of named constant values with an `int` variable. A C enumeration variable occupies 16 bits of memory. You define all integer operations on them. The C compiler provides no range checking, so an enumeration variable can hold a value not represented in the enumeration.

A TNS C routine can share an enumeration variable with TAL routines. A TAL routine cannot access the enumeration variables, but it can declare LITERALs for readability. For example:

```
TAL Code                          C Code
```

```
LITERAL no = 0,               enum choice {no = 0,
        yes = 3,                yes = 3,
        maybe = 4;              maybe = 4 };
BLOCK answer;                 enum choice ANSWER;
  INT answer_var;
  END BLOCK;
```

A TNS C routine can pass enumeration parameters to TAL routines, placing the actual value in a TAL INT variable. For example:

```
TAL Code                         C Code

LITERAL no = 0,               enum choice {no = 0,
        yes = 3,                yes = 3,
        maybe = 4;              maybe = 4 };
                              enum choice answer;

PROC tal_proc (n);
    INT n;                    _tal void TAL_PROC (short);
  BEGIN
  !Lots of code               main ()
  IF n = yes THEN ... ;       {
  !Lots of code                 answer = yes;
  END;                          TAL_PROC (answer);
                                /* lots of code */
                              }
```

Bit-Field Manipulation

You can manipulate bit fields in both TAL and C.

In TAL, you can use either:

- Built-in bit-extraction and bit-deposit operations
- Bit-wise operators LAND and LOR

In C, you can use either:

- Bit-wise operators & (and) and | (or)
- Defines

This TAL bit-deposit operation and C bit-wise operation are equivalent:

```
TAL Code                         C Code

INT x := -1;                  short a = -1;
INT y := 0;                   short b = 0;
                              short temp = 0;
PROC example;
  BEGIN                       void example ()
  y.<0:2> := x.<10:12>;       {
  END;                           /* you can combine these */
                                 /* with wider margins */
                                 temp = a & 070;
                                 temp = temp << 10;
                                 b = (b & 017777)|temp;
                               }
```

Bit extractions and bit deposits are not portable to future software platforms.

UNSIGNED Variables and Packed Bit Fields

In general, TAL UNSIGNED simple variables in structures are compatible with C unsigned packed bit fields (which appear only in structures). You cannot, however, pass C bit fields as reference parameters to TAL routines.

This UNSIGNED variables and C unsigned bit fields are compatible:

```
TAL Code                         C Code

STRUCT stuffed (*);           struct stuffed;
  BEGIN                       {
  INT x;                        int x;
  UNSIGNED(1) a;                unsigned a : 1;
  UNSIGNED(5) b;                unsigned b : 5;
  UNSIGNED(3) c;                unsigned c : 3;
  UNSIGNED(4) d;                unsigned d : 4;
  UNSIGNED(9) e;                unsigned e : 9;
  UNSIGNED(2) f;                unsigned f : 2;
  END;                        };
STRUCT packed (stuffed);      struct stuffed PACKED;
```

If the `WIDE` pragma is not specified, the TNS C compiler normally packs adjacent bit fields in a 16-bit word. If the `WIDE` pragma is specified, the TNS C compiler normally packs adjacent bit fields in a 32-bit word.

TAL UNSIGNED(1–16) and C bit fields of like size are compatible. TAL UNSIGNED(17–31) and C bit fields of like size are compatible.

The TAL compiler always packs adjacent UNSIGNED simple variables in 16-bit words:

- It starts the first UNSIGNED variable on a word boundary.
- It packs each successive UNSIGNED variable in the remaining bits of the same word as the preceding UNSIGNED variable if:
    - The variable contains 1 to 16 bits and fits in the same word
    - The variable contains 17 to 31 bits and fits in the same word plus the next word
- If an UNSIGNED variable does not fit in the same word or double word, the compiler starts the variable on the next word boundary.

The operator you use determines whether UNSIGNED values are signed or unsigned:

```
UNSIGNED(3) x;                !TAL code
UNSIGNED(3) y;
IF x + y ... ;                !Signed operation
IF x '+' y ... ;              !Unsigned operation
```

UNSIGNED arrays that contain 8-bit or 16-bit elements are compatible with C arrays that contain elements of like size. UNSIGNED arrays that contain 1-bit, 2-bit, or 4-bit elements are incompatible with C arrays.

TAL Routines as Parameters to TNS C

You can call TNS C routines and pass TAL routines as parameters. You can pass any TAL routine except EXTENSIBLE or VARIABLE routines as parameters.

A passed TAL routine can access the routine’s local variables and global TAL variables. The passed routine can contain subprocedures, but they cannot be passed as parameters.

If you call a large-memory-module TNS C routine, the EXTERNAL procedure declaration for the TNS C routine must specify the PROC(32) parameter type in the parameter declaration. When you pass the PROC(32) parameter to the TNS C routine, the compiler passes a 32-bit address that contains PEP and map information in the high-order word and a zero in the low-order word.

If you call a small-memory-module TNS C routine, the EXTERNAL procedure declaration for the TNS C routine must specify the PROC parameter type in the parameter declaration. When you pass the PROC parameter to the TNS C routine, the compiler passes a 16-bit address that contains PEP and map information.

In this example, a large-memory-model TNS C module contains C_FUNC, which expects a TAL procedure as a parameter. The TAL module contains:

- An EXTERNAL procedure declaration for C_FUNC
- TAL_PARAM_PROC, a routine to be passed as a parameter to C_FUNC
- TAL_CALLER, a routine that calls C_FUNC and passes TAL_PARAM_PROC as a parameter

C Module

```
 /* C function that accepts TAL routine as a parameter */

void C_FUNC (short (*F) (short n))
{
  short j;
  j = (*F)(2);
  /* lots of code */
}
```

TAL Module

```
PROC c_func (x) LANGUAGE C;  !EXTERNAL procedure declaration
                             ! for C routine to be called
    INT PROC(32) x;          !Parameter declaration
  EXTERNAL;
INT PROC tal_param_proc (f); !Procedure to be passed as a
    INT f;                   ! parameter to C_FUNC
  BEGIN
  RETURN f;
  END;
PROC tal_caller;             !Procedure that calls C_FUNC
  BEGIN                      ! and passes TAL_PARAM_PROC
  !Lots of code
  CALL c_func (tal_param_proc);
  !Lots of code
  END;
PROC m MAIN;
  BEGIN
  CALL tal_caller;
  END;
```

TNS C Routines as Parameters to TAL

You can call TAL routines and pass TNS C routines as parameters. You can call a TAL entry-point identifier as if it were the routine identifier. TNS C routines cannot be nested.

When a called TAL routine in turn calls a TNS C routine received as a parameter, the TAL routine assumes that all required parameters of the TNS C routine are value parameters. The TAL compiler has no way of checking the number, type, or passing method expected by the TNS C routine. If the TNS C routine requires a reference parameter, the TAL routine must explicitly pass the address by using:

- The @ operator for a small-memory-model parameter
- The $XADR standard function for a large-memory-model parameter

In this example, a TNS C large-memory-model module contains TNS C routine `C_PARAM_FUNC`, which is to be passed as a parameter. The TAL module contains:

- An EXTERNAL procedure declaration for C_PARAM_FUNC
- TAL_PROC, which expects C_PARAM_FUNC as a parameter
- TAL_CALLER, which calls TAL_PROC and passes C_PARAM_FUNC as a parameter

TAL Module

```
INT i;
STRING .EXT s[0:9];
PROC c_param_func (i, s)    !EXTERNAL procedure declaration
  LANGUAGE C;               ! for C routine expected as
    INT i;                  ! a parameter
    STRING .EXT s;          !Extended indirection for large-
  EXTERNAL;                 ! memory-model
PROC tal_proc (x);          !TAL routine that expects
    PROC(32) x;             ! a large-memory-model C routine
  BEGIN                     ! as a parameter
  CALL x (i, $XADR (s));
  END;
PROC tal_caller;
  BEGIN
  CALL tal_proc (c_param_func);
  END;
PROC m main;
  BEGIN
  CALL tal_caller;
  END;
```

C Module

```
void C_PARAM_FUNC (short i, char * s)
{                           /* C routine to be passed as */
                            /* a parameter to TAL_PROC   */
}
```

When you pass a large-memory-model TNS C routine as a parameter, the compiler passes a 32-bit address that contains PEP and map information in the high-order word and a zero in the low-order word. When you pass a small-memory-model TNS C routine as a parameter, the compiler passes a 16-bit address that contains PEP and map information.

Extended Data Segments

In addition to the user data segment, you can store data in:

- The automatic extended data segment
- One or more explicit extended data segments

You should use only the automatic extended data segment if possible. You should not mix the two approaches. If you must use explicit segments and the automatic segment, however, follow guidelines 4 and 11 in **Explicit Extended Data Segments** on page 136.

NOTE:

There are two types of extended data segments: flat segments and selectable segments. This section uses the term “extended data segment” to refer to a selectable segment. It does not describe flat segments, although much of the information is the same for flat segments. Flat segments provide many benefits over selectable segments. Flat segments are easier to program, provide better performance, and allow access to more virtual memory than selectable segments. For information on using flat segments, see the *Guardian Programmer’s Guide*.

Automatic Extended Data Segment

The TAL compiler allocates the automatic extended data segment when a TAL program declares arrays or structures that have extended indirection.

The TNS C compiler allocates the automatic extended data segment for all large‑memory-model modules.

Explicit Extended Data Segments

TAL modules and large-memory-model TNS C modules can allocate and deallocate extended data segments explicitly. Since the advent of the automatic extended data segment, however, programs usually need not use explicit extended data segments. The information in this subsection is provided to support existing programs.

To create and use an explicit extended segment, you call system procedures. You can allocate and manage as many explicit extended segments as you need, but you can use only one explicit extended segment at a time. You can access data in an explicit extended segment only by using extended pointers. The compiler allocates memory space for the extended pointers you declare. You must manage allocation of the data yourself.

Here are guidelines for using explicit extended data segments:

1. Declare an extended pointer for the base address of the explicit extended segment.
2. To allocate an explicit extended data segment and obtain the segment base address, call SEGMENT_ALLOCATE_.
3. To make the explicit extended segment the current segment, call SEGMENT_USE_.
4. If the automatic extended data segment is already in place, SEGMENT_USE_ returns the automatic extended data segment’s number. Save this number so you can later access the automatic extended data segment again.
5. C requires special treatment for SEGMENT_USE_, which returns a value and sets the condition code.
6. You must keep track of addresses stored in extended pointers. When storing addresses into subsequent pointers, you must allow space for preceding data items.
7. To refer to data in the current segment, call READX or WRITEX.
8. To move data between explicit extended data segments, call MOVEX.
9. To manage large blocks of memory, call DEFINEPOOL, GETPOOL, and PUTPOOL.
10. To determine the size of a segment, call SEGMENT_GETINFO_.

11. To access data in the automatic extended data segment, call SEGMENT_USE_ and restore the segment number that you saved at step 4.
12. To delete an explicit extended data segment that you no longer need, call SEGMENT_DEALLOCATE_.

For information on using these system procedures, see the *Guardian Programmer's Guide* and the *Guardian Procedure Calls Reference Manual*.

If you do not restore the automatic extended data segment before you manipulate data in it, any of these actions can result:

- An assignment statement is beyond the segment’s memory limit and causes a trap.
- All assignments within range occur in the hardware base and limit registers of the automatic extended segment. Data in the currently active extended data segment is overwritten. The error is undetected until you discover the discrepancy at a later time.
- The TNS C code runs until it accesses an invalid address or accesses an inaccessible library routine.
- You get the wrong data from valid addresses in the explicit extended data segment.

In this example, a large-memory-model TNS C routine calls a TAL routine that manipulates data in an explicit extended data segment and then restores the automatic extended data segment. When control returns to the TNS C routine, it manipulates data in the restored automatic extended data segment:

TAL Code

```
INT .EXT array[0:10];        !Allocated in the automatic
                             ! extended data segment ID 1024D
INT .EXT arr_ptr;
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS0 (
?     PROCESS_DEBUG_, DEFINEPOOL, GETPOOL,
?     SEGMENT_ALLOCATE_, SEGMENT_USE_, SEGMENT_DEALLOCATE_)
?POPLIST
PROC might_lose_seg;
  BEGIN
  INT status := 0;
  INT old_seg := 0;
  INT new_seg := 100;
  INT(32) seg_len := %2000D; !1024D
  array[0] := 10;         !Do work in automatic segment
  status := SEGMENT_ALLOCATE_ (
                 new_seg, seg_len, , , , , arr_ptr);
             !Allocate an explicit extended data segment;
             !store segment base address in ARR_PTR
  IF status <> 0 THEN CALL PROCESS_DEBUG_;
  Status := SEGMENT_USE_ (new_seg, old_seg, arr_ptr);
             !Make the explicit extended data segment current
  IF status <> 0 THEN CALL PROCESS_DEBUG_;
  !Use DEFINEPOOL, GETPOOL to retrieve a block in the
  ! explicit extended data segment.

  arr_ptr[2] := 10;        !Do some work in the segment.
  !When you no longer need the explicit extended data
  ! segment, call SEGMENT_DEALLOCATE_.
  status := SEGMENT_USE_ (old_seg);
             !Restore the automatic extended data segment
  IF status <> 0 THEN CALL PROCESS_DEBUG_;
  END;
```

C Code

```
#pragma symbols, inspect, strict
short arr[10];
char sarr[10];
char *s;
_tal void MIGHT_LOSE_SEG (void);
main ()
{
   s = &sarr[0];
   *s = 'A';
   arr[0] = 10;
   MIGHT_LOSE_SEG ();   /* Call TAL routine, which uses the*/
                        /* explicit extended data segment */
/* next two statements depend on the automatic extended */
/* data segment being restored after the call to TAL */
   sarr[1] = *s;
   arr[1] = arr[0] + 5;
}
```

Interfacing to TNS COBOL

Your TNS C/C++ programs can call functions written in TNS COBOL. The general procedure consists of these steps:

- Use the TNS COBOL compiler (COBOL85 in the Guardian environment or `cobol` in the OSS environment) to compile the COBOL function.
- Use the TNS C/C++ compiler (named `c`) to compile the C/C++ program.
- Use Binder to link the two object files and create the executable.

C Program that Calls a COBOL Function on page 138 shows the contents of an example TNS C program, named TESTC, that calls a COBOL function.

COBOL Function Called by a C Program on page 139 shows the contents of the COBOL function, named TESTCOB.

Include File (Prototype Function) on page 139 shows the contents of the header file used in this example, named COBINCLH.

Binder File on page 140 shows the contents of the Binder file, named BINDIN.

C Program that Calls a COBOL Function

```
/* TESTC */
#pragma inspect, symbols

#include <stdioh>  nolist
#include "COBINCLH"

short main (void)
{
  short ds;
  long  dl;
  char  *tx  = "Displayed in COBOL.";
  ds = 100;
  dl = 40000;
  XCOBFUNC (tx, &ds, &dl);
  printf ("I'm Back in C and program is ending.\n");
}
```

For examples showing TNS/R native C calling a COBOL function in the Guardian and OSS environments, see **Interfacing to Native COBOL** on page 162.

COBOL Function Called by a C Program

```
?env common
?SYMBOLS
 IDENTIFICATION DIVISION.
 PROGRAM-ID. XCOBFUNC.
 AUTHOR. ETREJO.
 DATE-WRITTEN. 7/25/00.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
   SOURCE-COMPUTER. TANDEM-K2006.
   OBJECT-COMPUTER. TANDEM-K2006.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 D-RESULT              PIC S9(09) COMP.
 LINKAGE SECTION.
 77 D-STRING              PIC X(20).
 77 D-SHORT               NATIVE-2.
 77 D-LARGE               NATIVE-4.

 PROCEDURE DIVISION USING D-STRING, D-SHORT, D-LARGE.
 000-INIT.
 CONFIGURATION SECTION.
   SOURCE-COMPUTER. TANDEM-K2006.
   OBJECT-COMPUTER. TANDEM-K2006.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 D-RESULT              PIC S9(09) COMP.
 LINKAGE SECTION.
 77 D-STRING              PIC X(20).
 77 D-SHORT               NATIVE-2.
 77 D-LARGE               NATIVE-4.
 PROCEDURE DIVISION USING D-STRING, D-SHORT, D-LARGE.
 000-INIT.
     DISPLAY "I AM DOING COBOL NOW".
     COMPUTE D-RESULT = D-LARGE / D-SHORT.
     DISPLAY "D-STRING = " D-STRING.
     DISPLAY "D-LARGE  = " D-LARGE.
     DISPLAY "D-SHORT  = " D-SHORT.
     DISPLAY "D-RESULT = " D-RESULT.
     DISPLAY "LEAVING COBOL PROGRAM NOW".
     EXIT-PROGRAM.
```

Include File (Prototype Function)

```
/* COBINCLH */
/* Following is the new way to declare extern COBOL calls */
void XCOBFUNC (char *, short *, long *);
#pragma FUNCTION XCOBFUNC (cobol)
```

```
select check parameter strong
add * from testco
add * from testcobo
select search $system.system.cwide
set heap 64
build testexe
```

To compile the programs in the Guardian environment:

1. Compile the COBOL function named `testcob` using the TNS COBOL compiler:

   ```
   cobol85 /in testcob / testcobo
   ```

2. Compile the C program named `testc` using the TNS C compiler:

   ```
   c / in testc/ testco
   ```

3. Link the object files using Binder and the file `BINDIN`:

   ```
   bind / in bindin, out listfile /
   ```

4. To run the executable named `testexe`:

   ```
   run testexe

   I AM DOING COBOL NOW
   D-STRING = Displayed in COBOL.
   D-LARGE  = 0000040000
   D-SHORT  = 00100
   D-RESULT = 000000400
   LEAVING COBOL PROGRAM NOW
   I'm Back in C and program is ending
   ```

Mixed-Language Programming for TNS/R, TNS/E, and TNS/X Native Programs

This chapter describes the Common Run-Time Environment (CRE) and the interface declarations that are necessary to interface from native C or C++ to other programming languages. The CRE and the interface declarations are HPE features for NonStop systems that are not available in standard ISO/ANSI C.

You can write native mixed-language programs targeted for the NonStop environment. Programs can contain routines written in native C, C++, pTAL, and native COBOL. The main() function in a native mixed-language program can be written in native C or C++, native COBOL, but not pTAL.

NOTE: TNS/R native objects cannot be linked with TNS/E and TNS/X native objects; nor you can link TNS/E objects with TNS/X objects.

Introducing the CRE

The Common Run-Time Environment (CRE) is a set of services that support mixed-language programs. The CRE library is a collection of routines that implement the CRE. The CRE library enables the language-specific run-time libraries to coexist peacefully with each other. User routines and run-time libraries call CRE library routines to access shared resources managed by the CRE, such as the standard files (input, output, and log) and the user heap, regardless of language.

The CRE does not support all possible operations. For example, the CRE supports file sharing only for the three standard files: standard input, standard output, and standard log. The language-specific run-time libraries access all other files by calling Guardian system procedures directly, whether or not a program uses the CRE.

All native mixed-language programs run in the CRE environment. Existing TAL code that is converted to pTAL must also be converted to run in the CRE. For more details on writing programs that use the services provided by the CRE, see the *Common Run-Time Environment (CRE) Programmer's Guide*.

Using Standard Files in Mixed-language Programs

In a mixed-language program, if a C function is to be the main function, it should be compiled with the `NOSTDFILES` pragma to keep it from automatically opening the three standard C files: `stdin`, `stdout`, and `stderr`.

If the main routine is not written in C, the three standard C files will not be automatically opened. If you want any or all the standard files to be opened for C, you must explicitly open them by calling the `fopen_std_file()` function.

Declaring External Routines

Your native C and C++ programs can call procedures written in COBOL and pTAL, in addition to procedures written in an unspecified language type. You cannot mix TNS and native mode languages.

All external procedures must be declared. When interfacing to pTAL or an unspecified language, you must declare the external procedure appropriately because lexical and operational features differ between C or C++ and other languages. There are two ways to declare an external function: using a function prototype and a FUNCTION pragma (the preferred method), and using an interface declaration (the traditional method).

Writing Interface Declarations

The interface declaration is an HPE extension of the function declaration as defined by ISO/ANSI C. The interface declaration indicates the correct language or indicates unspecified if the language is unknown. If no language attribute specifier is provided, the external procedure is assumed to be written in the same language as the compilation.

After providing an interface declaration, your C or C++ program uses normal function calls to access the procedure written in the other language. However, remember that these calls cross language boundaries; therefore, there are restrictions beyond those of normal C function calls.

The interface declaration enables you to declare COBOL procedures and most types of pTAL procedures, including:

- pTAL procedures defined with the VARIABLE or EXTENSIBLE attribute
- Procedures whose names are not valid C identifiers
- pTAL procedures that do not return a value but do return a condition code

However, your C or C++ program cannot directly call a pTAL procedure that returns both a value and a condition code. The subsection **pTAL Procedures That You Cannot Call Directly** on page 149 presents techniques that enable your C or C++ program to access pTAL procedures that fall into this category.

Using a Function Prototype and a FUNCTION Pragma

The recommended method for declaring an external routine is to use a standard function prototype followed later by a FUNCTION pragma.

For more details on syntax, see **FUNCTION** on page 239 and for illustrations of FUNCTION pragmas used in interface declarations, see **Examples** on page 143.

Using an Interface Declaration

The traditional method for declaring external routines is to use an interface declaration in which you declare the procedure as if it were a C function, except that you include:

- A language attribute specifier (`_cobol, _tal or _unspecified`) to identify the language of the external procedure. For native C and C++, `_tal` denotes the pTAL language on TNS/R, TNS/E, and TNS/X systems.
- An `_alias` attribute specifier to assign the external name, or rather the name as known to the other language.

The syntax for these older-style interface declarations is described in **Attribute Specifier** on page 61. However, it is recommended that you use the `FUNCTION` pragma to declare external routines.

Usage Guidelines

- Procedure names

  When you specify the C name of a non-C procedure, you should use the non-C name of the procedure if it is a valid C identifier. If the name is not a valid C identifier because it includes circumflexes (^) or hyphens (-), you must use the `_alias` attribute in the interface declaration.

To create a C function name that resembles the pTAL procedure name, you can substitute an underscore (_) for every occurrence of the circumflex or hyphen in the pTAL name.

- Return value types

  For procedures that return a scalar value, you should declare the C counterpart to the pTAL scalar type as the procedure type in the interface declaration. **Compatible pTAL and C Data Types** shows the C counterparts to pTAL scalar types.

  For procedures that return a condition code, you should declare `_cc_status` as the procedure's return type in the interface declaration. The `tal.h` header file contains six macros to interrogate the results of the procedure declared with the `_cc_status` type specifier:

  ```
  #define _status_lt(x) ((x) <  0)
  #define _status_le(x) ((x) <= 0)
  #define _status_eq(x) ((x) == 0)
  #define _status_ge(x) ((x) >= 0)
  #define _status_gt(x) ((x) >  0)
  #define _status_ne(x) ((x) != 0)
  ```

  Before you can use these macros, you must include the `tal.h` header.

  Note that you should avoid designing pTAL procedures that return a condition code because that is an outdated programming practice. Guardian system procedures must retain this interface for reasons of compatibility.

- Parameter types

  When creating an interface declaration for a non-C procedure, you must ensure that the declared C type of a parameter matches the defined type of that parameter. The C compiler cannot perform this task automatically, because it does not have direct access to each language's definition. The C compiler does, however, check that the type of an argument in a call to a non-C procedure matches its corresponding parameter's type. Ensuring that the C and non-C types of a scalar parameter match is a simple task because most scalar types have direct counterparts in C, as shown in **Data Type Correspondence** on page 559.

- When you invoke a COBOL procedure, COBOL always returns `void` because the COBOL language has no way to declare a COBOL program to be a function. Therefore, you must always specify COBOL routines with type `void`.
- Native COBOL programs use 32-bit addressing for all data items in the Extended-Storage Section and all data items in the Linkage Section that are not described as having ACCESS MODE of STANDARD.
- If the pTAL definition specifies the EXTENSIBLE attribute, the interface declaration must specify `_extensible`.
- If the pTAL definition specifies the VARIABLE attribute, the interface declaration must specify `_variable`.

Examples

1. This example, taken from the `stdlib.h` header file, shows the preferred style for declaring an external routine (in this case, for the CRE_ASSIGN_MAXORDINAL_ procedure). The declaration consists of a standard function prototype and a `FUNCTION` pragma:

   ```
   void get_max_assign_msg_ordinal (void);
   ...
   ```

```
#pragma function get_max_assign_msg_ordinal
(alias(“CRE_ASSIGN_MAXORDINAL_”), extensible, tal)
```

The example above is equivalent to this interface declaration:

```
_extensible _tal _alias (“CRE_ASSIGN_MAXORDINAL_”)
get_max_assign_msg_ordinal (void);
```

2. This example, taken from the `cextdecs` header, shows the interface declaration for the CHANGELIST system procedure. Notice that identifier names in the parameter type list are optional, because C cares only about the types of the parameters:

```
_tal _variable _cc_status CHANGELIST(short, short, short);
```

3. This example shows you how to use the `_cc_status` type specifier while also using a function prototype and a FUNCTION pragma to declare an external procedure.

   Notice that the variable `c_code` is declared as type `_cc_status` so that it can be compared against the condition code that is contained in the `_status_eq()` macro:

```
#include <tal.h>  /* defines _cc_status and _status_eq */

_cc_status C_GETPOOL (short *pool_head,
                      long   block_size,
                      long*  block_addr);
#pragma function C_GETPOOL (tal, alias("C^GETPOOL"))

short *pool_alloc ( short *my_pool, long size )
{
   _cc_status c_code;
   long       blk_addr;

   c_code = C_GETPOOL(my_pool, size, &blk_addr);
   if (_status_eq(c_code))
      return( (short *)blk_addr );
   else
      return( NULL );
}
```

4. This function prototype and FUNCTION pragma:

```
void segment_allocate (short, long, short *, short);
...
#pragma function segment_allocate (tal, alias
(“SEGMENT_ALLOCATE_”), variable)
```

   is equivalent to this interface declaration:

```
_tal _variable short SEGMENT_ALLOCATE_ (short, long,
                                       short *, short);
```

5. This declarations of external routines are written in both of the styles described in this section for calling pTAL procedures:

```
void c_name (short *);
#pragma function c_name (tal, alias (“tal^name”))
/* function declaration...*/
_tal _variable _cc_status SEGMENT_DEALLOCATE_ (short, short);

_tal _variable _cc_status READ (short, short *, short,
                                    short *, long);
_tal _extensible _cc_status READX (short, char *,
                                      short, short *, long);
```

Considerations When Interfacing to pTAL

This subsection provides guidelines for writing programs composed of pTAL and native C or C++ modules. All the discussion that pertains to native C also pertains to native C++, unless otherwise noted. The discussion assumes that you have a working knowledge of pTAL and C, and are familiar with the contents of the:

- *pTAL Conversion Guide*
- *pTAL Reference Manual*

Using Identifiers

pTAL and C identifiers differ:

- pTAL and C have independent sets of reserved keywords.
- pTAL identifiers can include circumflexes (^); C identifiers cannot.
- The C language is case-sensitive; the pTAL language is not case-sensitive.

To declare variable identifiers that satisfy both compilers:

- Avoid using reserved keywords in either language as identifiers.
- Specify pTAL identifiers without circumflexes.
- Specify C identifiers in uppercase.
- For C or C++, you can use the _alias attribute specifier to describe the name of an external pTAL routine that does not have a valid C name.

You can declare pTAL-only, or C-only routine identifiers and satisfy both compilers by using the public name option in:

- Interface declarations in C
- EXTERNAL procedure declarations in pTAL

In G-series Inspect or H-series Native Inspect sessions:

- Use uppercase for pTAL identifiers
- Use the given case for C identifiers

Matching Data Types

Use data types that are compatible between languages for:

- Shared global variables
- Formal or actual parameters
- Function return values

Table 22: Compatible pTAL and C Data Types

pTAL Declaration		pTAL Data Type	C Data Type
Direct data			
STRING	i;	STRING	`char`
INT	i;	INT (signed)	`short`
INT (32)	j;	INT(32)	`int`
FIXED	f;	FIXED(0)	`long long`
REAL	r;	REAL	`float`
REAL(64)	s;	REAL(64)	`double`
Indirect using 16-bit pointers using the pTAL or TAL model			
STRING	.s;	STRING	`char *`
INT	.i;	INT	`short *`
INT (32)	.j;	INT(32)	`int *`
FIXED	.f;	FIXED(0)	`long long *`
REAL	.r;	REAL	`float *`
REAL(64)	.s;	REAL(64)	`double *`
Indirect using 32-bit pointers			
STRING	.EXT s;	STRING	`char *`
INT	.EXT i;	INT	`short *`
INT (32)	.EXT j;	INT(32)	`int *`
FIXED	.EXT f;	FIXED(0)	`long long *`
REAL	.EXT r;	REAL	`float *`
REAL(64)	.EXT s;	REAL(64)	`double *`

Incompatibilities between pTAL and C data types include:

- pTAL has no numeric data type that is compatible with C `unsigned long` type.
- pTAL UNSIGNED is not compatible with the C `unsigned short` type. pTAL UNSIGNED(16), INT, or INT(16) can represent signed or unsigned values.

For more details on C and pTAL data types, see **Parameters and Variables** on page 155.

Calling C Routines From pTAL Modules

A pTAL module must include an EXTERNAL procedure declaration for each C routine to be called. This pTAL code shows the EXTERNAL procedure declaration for C_FUNC and a routine that calls C_FUNC. ARRAY is declared with .EXT, because C_FUNC uses the large-memory model:

```
pTAL Code                       C Code

INT status := 0;             short C_FUNC(char *str)
STRING .EXT array[0:4];      {
                                *str = 'A';
INT PROC c_func (a)             str[2] = 'C';
  LANGUAGE C;                   return 1;
    STRING .EXT a;           }
EXTERNAL;
PROC example MAIN;
  BEGIN
  array[2] := "B";
  status := c_func (array);
  array[1] := "B";
  END;
```

C Routines That You Cannot Call Directly

pTAL procedures cannot call a C routine that:

- Passes a struct or union parameter by value
- Returns a struct or union parameter by value
- Uses C-style variable argument list (...)

Calling pTAL Routines From C Modules

A C module has full access to the C run-time library even if the C module does not contain the main() routine, but the C module might need to explicitly initialize the C run time library.

When you code C modules that call pTAL routines:

- Include an interface declaration or a function prototype and FUNCTION pragma for each pTAL routine to be called.
- If a called pTAL routine sets the condition code, include the `talh` header file.
- If a called routine is a system procedure, include the `cextdecs` header file.

It is recommended that you declare an external routine by using a function prototype and FUNCTION pragma instead of an interface declaration. For more details, see **Using a Function Prototype and a FUNCTION Pragma** on page 142. In C, an interface declaration is comparable to an EXTERNAL procedure declaration in pTAL. For more details, see **Using an Interface Declaration** on page 142.

Specifying a FUNCTION Pragma

To declare an external routine by using a function prototype and a FUNCTION pragma, include:

- The keyword _tal
- The variable or extensible attribute, if any, of the pTAL routine
- A public name if the pTAL identifier is not a valid C identifier

Specifying an Interface Declaration

To declare an external routine by using an interface declaration, include:

- The keyword _tal
- The variable or extensible attribute, if any, of the pTAL routine
- The data type of the return value, if any, of the pTAL routine
- An identifier for the pTAL routine
- A public name if the pTAL identifier is not a valid C identifier
- A parameter-type list or, if no parameters, the keyword void

The return type value can be any of these:

Return Type Value	Meaning
void	The pTAL routine does not return a value.
fundamental-type	The pTAL routine returns a value. Specify one of, or a pointer to one of, the character, integer, or floating-point types.
_cc_status	The pTAL routine does not return a value but does set a condition code. The tal.h header file contains six macros that interrogate this condition code: _status_lt(x), _status_le(x), _status_eq(x), _status_ge(x), _status_gt(x), and _status_ne(x).

For information on calling pTAL routines that both return a value and set a condition code, see **pTAL Procedures That You Cannot Call Directly** on page 149.

For examples of interface declarations for calling pTAL routines, see **Examples** on page 143.

For corresponding pTAL source for these first four functions, see the *Guardian Procedure Calls Reference Manual* or the EXTDECS file.

After specifying an interface declaration, use the normal C routine call to access the pTAL routine. This example shows a C module that calls an pTAL routine:

C Code

```
#include <stdioh> nolist
short arr[5];        /*stored in extended segment */
_tal _alias ("TAL^NAME") void C_Name (short *);

void func1 (short *xarr)
{
  C_Name (xarr);
  printf ("xarr[2] after TAL = %d", xarr[2]);
}

main ()
{
  arr[4] = 8;
  func1 (arr);
}
```

pTAL Code

```
PROC TAL^NAME (a);
    INT .EXT a;          !32-bit pointer
  BEGIN
  a[2] := 10;
  END;
```

pTAL Procedures That You Cannot Call Directly

Your C program cannot directly call a pTAL procedure that returns both a value and a condition code. To access a pTAL procedure that returns both a value and a condition code, you must write a “jacket” procedure in pTAL that is directly callable by your C program. Define this jacket procedure so that it:

- Passes arguments from C calls through to the pTAL procedure unchanged
- Returns the pTAL procedures return value indirectly through a pointer parameter
- Returns the condition code using the function-return type

  ```
  _cc_status
  ```

To illustrate this technique, this example defines a jacket procedure for the GETPOOL system procedure. First, here is the pTAL source module that defines the jacket routine:

```
?SOURCE EXTDECS(GETPOOL);

INT PROC C^GETPOOL(pool^head, block^size, block^addr);
  INT .EXT pool^head;     ! Address of pool head
  INT(32) block^size;     ! Block size
  INT(32) .block^addr;    ! Block address

BEGIN
  ! Call GETPOOL, storing return value in block^addr
  block^addr := GETPOOL(pool^head, block^size);
  if < then
     return -1
  else if > then return 1 else return 0;
RETURN;  ! No return value here
END;
```

The definition of C^GETPOOL declares the same parameters as the GETPOOL system procedure, with the addition of block^addr. This additional parameter is a pointer to the return type of GETPOOL, which is INT(32). Note that the RETURN statement specifies no return value and that it immediately follows the GETPOOL call, therefore ensuring that the condition-code is unchanged.

Here is the interface declaration for C^GETPOOL as you would enter it in your C module:

```
_tal _cc_status _alias ("C^GETPOOL") C_GETPOOL
                                    (short *pool_head,
```

```
                                             long block_size,
                                             long *block_addr);
```

Once you have made this declaration, you can effectively call GETPOOL by calling the jacket routine C_GETPOOL; for example:

```
short *pool_alloc(long size)

{
  short c_code;
  long blk_addr;

  c_code = C_GETPOOL(my_pool,size,&blk_addr);
  if (_status_eq(c_code))
    return( (short *)blk_addr );
  else
    return( NULL );
}
```

Techniques other than this cannot be used on TNS/E and TNS/X systems. For more details, see the Appendix D of the *pTAL Reference Manual.*

Sharing Data

Using pointers to share data is easier and safer than trying to match declarations in both languages. Using pointers also eliminates problems associated with where the data is placed.

The default code generation for pointer dereferencing operations (REFALIGNED 8) expects the pointer to contain an address that satisfies the alignment requirements of the object being pointed to. For example, a 4-byte object should have an address that is a multiple of 4.

If the object is at an address that does not satisfy its alignment requirements, the default code generation will cause a compatibility trap. To avoid this, specify the REFALIGNED 2 C pragma or the REFALIGNED 2 pTAL directive on the pointer to the object. This will result in code generation that assumes that the dereferenced object is not properly aligned and will compensate for that.

To share data by using pointers, first decide whether the pTAL module or the C module is to declare the data:

- If the pTAL module is to declare the data, follow the guidelines in **Sharing pTAL Data With C Using Pointers** on page 151.
- If the C module is to declare the data, follow the guidelines in **Sharing C Data With pTAL Using Pointers** on page 152
- If both the pTAL and the C modules are to declare the data, follow the guidelines in **Sharing pTAL Data With C Using BLOCK Declarations** on page 153 or in **Sharing pTAL Global Data With C/C++ Using BLOCK Declarations** on page 153.

For 64-bit unsigned data, use the **FIELDALIGN** on page 232 pragma to select the data alignment that is appropriate for the compiler you use:

- `SHARED2` directs the TNS C, native C, and native C++ compilers to lay out components using the default TAL compiler alignment rules.
- `SHARED8` directs the TNS C, native C, and native C++ compilers to lay out components using the pTAL compiler

  `SHARED8` alignment rules.

`FIELDALIGN AUTO` is recommended when the data layout is target independent and not shared.

`FIELDALIGN PLATFORM` is used to share data across languages for the same hardware platform. It lays out components using the platform's C compiler `AUTO` rules. That is, `PLATFORM` directs the TNS C compiler to lay out components using the `SHARED2` alignment rules that are the `AUTO` rules for TNS platforms, and directs the native C and C++ compilers to lay out components using the `AUTO` alignment rules for their respective TNS/R, TNS/E, and TNS/X platforms.

Sharing pTAL Data With C Using Pointers

To share pTAL global data with C modules, follow these steps:

1. In the pTAL module, declare the data using C-compatible identifiers, data types, and alignments. (Alignments depend on byte or word addressing and variable layouts as described in **Parameters and Variables** on page 155.)
2. In the C module, declare pointers to the data, using pTAL-compatible data types.
3. In the C module, declare a routine to which pTAL can pass the addresses of the shared data.
4. In the C routine, initialize the pointers with the addresses sent by the pTAL module.
5. Use the pointers in the C module to access the pTAL data.

This example shows how to share pTAL data with a C module. The pTAL module passes to a C routine the addresses of two pTAL arrays. The C routine assigns the array addresses to C pointers.

C Code

```
short *c_int_ptr;         /* pointer to pTAL data */
char *c_char_ptr;            /* pointer to pTAL data */
short INIT_C_PTRS (short *tal_intptr, char *tal_strptr)
{                            /* called from pTAL */
  c_int_ptr = tal_intptr;
  c_char_ptr = tal_strptr;
  return 1;
}
/* Access the pTAL arrays by using the pointers */
```

pTAL Code

```
STRUCT rec (*);
  BEGIN
  INT x;
  STRING tal_str_array[0:9];
  END;

INT  tal_int_array[0:4];  !pTAL data to share with C
STRUCT  tal_struct (rec);    !pTAL data to share with C
INT status := -1;

INT PROC init_c_ptrs (tal_intptr, tal_strptr) LANGUAGE C;
    INT .EXT tal_intptr;
    STRING .EXT tal_strptr;
  EXTERNAL;
PROC tal_main MAIN;
  BEGIN
  status := init_c_ptrs
                 (tal_int_array, tal_struct.tal_str_array);
  !Do lots of work
  END;
```

Sharing C Data With pTAL Using Pointers

To share C global data with pTAL modules, follow these steps:

1. In the C module, declare the data using pTAL-compatible identifiers, data types, and alignments. (Alignments depend on byte or word addressing and variable layouts as described in **Parameters and Variables** on page 155.)

   C arrays and structures are automatically indirect.

2. In the pTAL module, declare pointers to the data, using C-compatible data types.
3. In the pTAL module, declare a routine to which C can pass the addresses of the shared data.
4. In the pTAL routine, initialize the pointers with the addresses sent by C.
5. Use the pointers in the pTAL module to access the C data.

This example shows how to share C data with an pTAL module. The C module passes the addresses of two C arrays to a pTAL routine. The pTAL routine assigns the array addresses to pTAL pointers.

C Code

```
#include <stdioh> nolist

short arr[5];         /* C data to share with pTAL */
char charr[5];          /* C data to share with pTAL */
_tal void INIT_TAL_PTRS (short *, char *)
_tal _alias ("tal^name") void C_Name (void);
void example_func( int *x )
{
  printf("x before TAL = %d\n", x[2] );
  C_Name( );
  printf("x after TAL = %d\n", x[2] );
}
main ()
{
  INIT_TAL_PTRS ( &arr[0], &charr[0] ); /* initialize ptrs */
    /* test pointer values */
  arr[0] = 8;
  example_func( arr );
  arr[2] = 18;
  charr[2] = 'B';
}
```

pTAL Code

```
INT .EXT tal_int_ptr;                  !Pointer to C data
STRING .EXT tal_char_ptr;                  !Pointer to C data
PROC init_tal_ptrs (c_addr1, c_addr2); !Called from C
    INT .EXT c_addr1;
    STRING .EXT c_addr2;
  BEGIN
  @tal_int_ptr := @c_addr1;
  @tal_char_ptr := @c_addr2;
  END;
PROC tal^name;
  BEGIN
  tal_int_ptr[0] := 10;
  tal_int_ptr[2] := 20;
  tal_char_ptr[2] := "A";
  END;
```

Sharing pTAL Data With C Using BLOCK Declarations

pTAL modules can share global data with C modules by declaring each shared variable in its own BLOCK declaration and giving both the variable and the BLOCK the same name. The C modules must also declare each shared variable; the layout of the variable must match in both the pTAL and C modules.

In this example, a pTAL module declares a variable within a BLOCK declaration, and the C module declares the equivalent variable:

```
pTAL Code                    C Code

NAME TAL_module;
BLOCK fred;
  INT .EXT fred;           int FRED;  /*all uppercase*/
  END BLOCK;
```

Because the preceding method requires that the layout of the corresponding pTAL and C declarations match, it is recommended that you share data by using pointers where possible.

Sharing pTAL Global Data With C/C++ Using BLOCK Declarations

The pTAL directive BLOCKGLOBALS can be used to interface pTAL global data declarations with C/C++. BLOCKGLOBALS directs the pTAL compiler to separate implicitly all data declarations into individual blocks. For example:

```
pTAL Code                      C/C++ Code

?BLOCKGLOBALS
?EXPORTGLOBALS

name x;

int i;                            extern short i;
int j;                            extern short j;
int k [0:2] = 'P' := [0, 1, 2];     extern const short k [3] := {0, 1, 2};

block b;
  int .ext p;                     extern short* p;
  int .ext s (template);          extern Template s;
end block;

block c;
  struct s1 (template);           extern Template s1
  int a [0:1];                    extern short a [2];
end block;
```

If you want variable definitions to be in the C++ source, remove the `extern` specifier and change the pTAL source to ?NOEXPORTGLOBALS.

In addition, if ?BLOCKGLOBALS is in effect and a data block contains one data declaring item, the name exported to C/C++ is the name of the block. This method makes it easy to change the name of a global declaration which cannot be described in C/C++ to a legal C/C++ name. With this method, you do not need to change all uses of the global declaration in the pTAL source. For pTAL programs, P-relative arrays allocate space in the read-only data area of process memory space. For example:

```
pTAL Code                      C/C++ Code

?BLOCKGLOBALS
```

```
?EXPORTGLOBALS

name x;

block b;
  int i^1;                         extern short b;
end block;

block c;
  int .i [0:9];                    extern short c [10];
end block;

block d;
  int j = 'P' := [1, 2, 3];        extern short d [3] := {1, 2, 3};
end block;

block b1;
                                   /* Names are not changed because there  */
                                   /* is more than one data declaring item */
                                   /* in the block.                        */
  int k;                           extern short k;
  int l;                           extern short l;
end block;

block b2;
                                   /* Names are not changed because there  */
                                   /* is more than one data declaring item */
                                   /* in the block.                        */
  int .m [0:9];                    extern short m [0:9];
  int n;                           extern short n;
end block;

block b3;
                                   /* Names are not changed because there  */
                                   /* is more than one data declaring item */
                                   /* in the block.                        */
  int o = 'P' := [1, 2, 3];        extern short o [3] := {1, 2, 3};
  int p;                           extern short p;
end block;

block b4;
                                   /* Names are not changed because there  */
                                   /* is more than one data declaring item */
                                   /* in the block.                        */
  int q = 'P' := [1, 2, 3];        extern short q [3] := {1, 2, 3};
  int .r [0:9];                    extern short r [10];
end block;

block b5;
                                   /* Names are not changed because there  */
                                   /* is more than one data declaring item */
                                   /* in the block.                        */
  int s = 'P' := [1, 2, 3];        extern short s [3] := {1, 2, 3};
  int t = 'P' := [1, 2, 3];        extern short t [3] := {1, 2, 3};
end block;

block e;
```

```
                                   /* Only one data declaring item. */
  int u;                           extern short e;
  int v = u;
end block;
```

If variable definitions are desired to be in the C++ source, remove the `extern` specifier and change the pTAL source to ?NOEXPORTGLOBALS.

Parameters and Variables

This subsection provides guidelines for declaring compatible pTAL and C variables and parameters. These guidelines supplement those given in **Sharing Data** on page 150. The topics discussed are:

- **STRING and char Variables** on page 155
- **Arrays** on page 155
- **Structures** on page 156
- **Multidimensional Arrays** on page 157
- **Arrays of Structures** on page 157
- **Redefinitions and Unions** on page 157
- **Pointers** on page 158
- **Enumeration Variables** on page 159
- **pTAL Routines as Parameters to C** on page 159
- **C Routines as Parameters to pTAL** on page 160

STRING and char Variables

pTAL STRING and C char simple variables each occupies one byte of a word. Share variables of type pTAL STRING and C char by using pointers or declare both as int.

Arrays

pTAL and C arrays differ:

Characteristic	pTAL Array	C Array
Lower bound	Any integer	Always zero
Dimensions	One dimension *	One or more dimensions
Direct or indirect	Direct or indirect	Indirect only
Byte or word addressing	STRING arrays and extended indirect arrays are byte addressed; all other arrays are word addressed	Native C and C++ only have byte addressing

* pTAL structures can emulate multidimensional C arrays, as discussed in **Multidimensional Arrays** on page 157.

To declare compatible pTAL and C arrays:

- Use data types and alignments that satisfy both compilers.
- Declare pTAL arrays that have a lower bound of 0.
- Declare one-dimensional C arrays.
- Declare indirect pTAL arrays.

These are compatible arrays in pTAL and native C:

```
pTAL Code                      C Code
INT .EXT robin[0:9];        short robin [10];
INT(32) .EXT gull[0:14];    long gull [15];
STRING .EXT grebe[0:9];        char grebe [10];
```

Structures

The FIELDALIGN C pragma controls the component layout of structures for compatibility between TNS and native structure layout and for compatibility with native mixed-language structure layout. Therefore, you need to:

- Use the FIELDALIGN SHARED2 pragma to share data between TNS programs and native programs.
- Use the FIELDALIGN SHARED8 pragma to share data between native C/C++ programs and pTAL programs that run on different platforms.

  SHARED8 requires that any filler needed to align fields properly be explicitly declared. The compiler issues a warning for improperly aligned fields in a SHARED8 structure. (You can also use the

  FIELDALIGN PLATFORM pragma to share data between native C/C++ and pTAL programs, but that data cannot be shared with any TNS programs.)

All pTAL and C structures begin on a 16-bit boundary. These are guidelines for sharing pTAL and C structures and passing them as parameters:

- Specify the same layout for corresponding pTAL and C structures.
- Specify compatible data types for each item of both structures.
- In pTAL, pass structures by reference.
- In C, use the & (ampersand) operator.
- In pTAL, a routine cannot return a structure as a return value or pass a struct or union by value.

This pTAL and C structures have compatible layouts:

```
pTAL Code                          C Code

STRUCT rec (*);                 struct birdname
  BEGIN                         {
  INT x;                          short x;
  STRING y[0:2];                  char y[3];
  END;                          } robin[10];
STRUCT .EXT robin(rec)[0:9];
```

This pTAL and C structures have compatible layouts:

```
pTAL Code                          C Code
STRUCT rec1 (*);                struct rec1
```

```
  BEGIN                         {
  STRING a, b, c;                 char a, b, c;
  END;                          };
```

This pTAL and C structures also have compatible layouts:

```
pTAL Code                          C Code
STRUCT rec2 (*);                struct rec2
  BEGIN                         {
  STRING e;                       char e;
  INT y;                          short y;
  STRING g;                       char g;
  END;                          };
```

Multidimensional Arrays

In C, you can declare multidimensional arrays. In pTAL, you can emulate multidimensional arrays by declaring structures that contain arrays.

Here is an example of multidimensional arrays in pTAL and native C:

```
pTAL Code                        C Code

STRUCT rec1 (*);
  BEGIN
  INT y[0:4];                    short cma[10][5];
  END;
STRUCT .EXT tma(rec1)[0:9];
!Sample access!                  /* sample access */
tma[8].y[3] := 100;              cma[8][3] = 100;
```

Arrays of Structures

If you specify bounds when you declare a pTAL structure, you create an array of structures. This pTAL and C arrays of structures are equivalent. Each declaration contains an array of ten structure occurrences:

```
pTAL Code                          C Code

STRUCT cell (*);                struct cell
  BEGIN                         {
  INT x;                          short x;
  STRING y;                       char y;
  END;                          };
STRUCT .EXT tcell(cell)[0:9];   struct cell ccell [10];
PROC honey (c);                 void JOANIE
    INT .EXT c (cell);          (struct cell *c);
EXTERNAL;
```

Redefinitions and Unions

Variant records are approximated by pTAL structure redefinitions and C unions.

An pTAL redefinition declares a structure item that uses the same memory location as an existing structure item. The existing structure item can be a simple variable, array, substructure, or pointer that:

- Begins on a word boundary
- Is at the same BEGIN-END level in the structure as the redefinition
- Is the same size or larger than the redefinition

A C union defines a set of variables that can have different data types and whose values alternatively share the same portion of memory. The size of a union is the size of its largest variable; the largest item need not come first.

Although a union typically begins on a word boundary, native C sometimes misaligns numeric data that is inside a union. In a union, a numeric data item (short, long, int, or long long) is misaligned if the data item is at an odd byte offset from the beginning of the union, and the union itself begins at an odd byte address. If possible, you should rearrange the data structure to ensure that unions begin on an even byte address (and therefore, NMC or CCOMP does not insert an implicit filler byte).

A union typically, but not always, begins on a word boundary. If FIELDALIGN SHARED2 is in effect, and if the union contains only character data, a union might begin on an odd-byte (that is, not be word-aligned).

Pointers

Pointers contain memory addresses of data. You must store an address into a pointer before you use it. In pTAL and C pointer declarations, you specify the data type of the data to which the pointer points. You must use pointers when sharing global variables. You can pass pointer contents by value between pTAL and C routines.

Differences between pTAL and C pointers include:

- pTAL structure pointers can point to a byte or word address.
- C structure pointers always point to a byte address.
- pTAL pointers are dereferenced implicitly.

Here are examples of pTAL and native C pointers:

```
pTAL Code                         C Code

STRUCT rec (*);                struct rec
  BEGIN                        {
  INT d;                       short d;
  INT .p (rec);                struct rec *p;
  END;                           };
BLOCK joe;
  INT .EXT joes (rec);         struct rec *JOE;
  END BLOCK;
PROC tonga (p);                void CALEDONIA
    INT .EXT p (rec);          (struct rec *p)
  BEGIN                        {
  !Lots of code                /* Lots of code */
  END;                           }
```

Each language can call the other, passing the address in the pointer by value:

```
pTAL Code                        C Code

CALL caledonia (joes);           TONGA (joe);
```

Here are examples of pTAL and C structure pointers (large-memory model) that implement a linked list:

```
pTAL Code                        C Code

STRUCT rec (*);               struct rec
  BEGIN                       {
  INT x;                         short x;
  INT .EXT strptr (rec);         struct rec *p;
END;                             };
STRUCT .EXT joe (rec);           struct rec joe;
PROC callme (param1);         void f1 (struct rec *param1);
    INT .EXT param1 (rec);
EXTERNAL;
```

Enumeration Variables

Using C enumeration variables, you can associate a group of named constant values with an `int` variable. A C enumeration variable occupies 16 bits of memory for TNS programs and 32 bits for native programs. You define all integer operations on them. The C compiler provides no range checking, so an enumeration variable can hold a value not represented in the enumeration.

A C routine can share an enumeration variable with pTAL routines. A pTAL routine cannot access the enumeration variables, but it can declare LITERALs for readability. For example:

```
pTAL Code                        C Code

LITERAL no = 0,                 typedef enum choice
        yes = 3,                 {  no = 0,
        maybe = 4;                  yes = 3,
                                    maybe = 4
BLOCK answer;                    }  choice;
  INT answer_var;
  END BLOCK;                     choice ANSWER;
```

A C routine can pass enumeration parameters to pTAL routines, placing the actual value in a pTAL INT variable. For example:

```
pTAL Code                        C Code

LITERAL no = 0,               enum choice {no = 0,
        yes = 3,                yes = 3,
        maybe = 4;               maybe = 4 };
                              enum choice answer;
PROC tal_proc (n);
    INT n;                    _tal void TAL_PROC (short);
  BEGIN
  !Lots of code               main ()
  IF n = yes THEN ... ;       {
  !Lots of code                 answer = yes;
  END;                          TAL_PROC (answer);
                                /* lots of code */
                              }
```

pTAL Routines as Parameters to C

You can call C routines and pass pTAL routines as parameters. You can pass any pTAL routine except EXTENSIBLE or VARIABLE routines as parameters.

A passed pTAL routine can access the routine’s local variables and global pTAL variables. The passed routine can contain subprocedures, but they cannot be passed as parameters.

In this example, a C module contains C_FUNC, which expects a pTAL procedure as a parameter. The pTAL module contains:

- An EXTERNAL procedure declaration for C_FUNC
- TAL_PARAM_PROC, a routine to be passed as a parameter to C_FUNC
- TAL_CALLER, a routine that calls C_FUNC and passes TAL_PARAM_PROC as a parameter

C Module

```
/* C function that accepts pTAL routine as parameter */
void C_FUNC (short (*F) (short n))
{
  short j;
  j = (*F)(2);
  /* lots of code */
}
```

pTAL Module

```
PROC c_func (x) LANGUAGE C;  !EXTERNAL procedure declaration
                             ! for C routine to be called
    INT PROC x;              !Parameter declaration
  EXTERNAL;

INT PROC tal_param_proc (f); !Procedure to be passed as a
    INT f;                   ! parameter to C_FUNC
  BEGIN
  RETURN f;
  END;

PROC tal_caller;             !Procedure that calls C_FUNC
  BEGIN                      ! and passes TAL_PARAM_PROC
  !Lots of code
  CALL c_func (tal_param_proc);
  !Lots of code
  END;

PROC m MAIN;
  BEGIN
  CALL tal_caller;
  END;
```

C Routines as Parameters to pTAL

You can call pTAL routines and pass C routines as parameters. You can call a pTAL entry-point identifier as if it were the routine identifier. C routines cannot be nested.

When a called pTAL routine in turn calls a C routine received as a parameter, the pTAL routine assumes that all required parameters of the C routine are value parameters. The pTAL compiler has no way of checking the number, type, or passing method expected by the C routine. If the C routine requires a reference parameter, the pTAL routine must explicitly pass the address by using the $XADR standard function for a parameter.

In this example, a C module contains C routine C_PARAM_FUNC, which is to be passed as a parameter. The pTAL module contains:

- An EXTERNAL procedure declaration for C_PARAM_FUNC
- TAL_PROC, which expects C_PARAM_FUNC as a parameter
- TAL_CALLER, which calls TAL_PROC and passes C_PARAM_FUNC as a parameter

pTAL Module

```
INT i;
STRING .EXT s[0:9];

PROC c_param_func (i, s)   !EXTERNAL procedure declaration
  LANGUAGE C;              ! for C routine expected as
    INT i;                 ! a parameter
    STRING .EXT s;         !Extended indirection for large-
  EXTERNAL;                   ! memory-model
PROC tal_proc (x);         !pTAL routine that expects
    PROC x;                ! a C routine as a parameter
  BEGIN
  CALL x (i, $XADR (s));
  END;
PROC tal_caller;
  BEGIN
  CALL tal_proc (c_param_func);
  END;
PROC m main;
  BEGIN
  CALL tal_caller;
  END;
```

C Module

```
void C_PARAM_FUNC (short i, char * s)
{                          /* C routine to be passed as */
                           /* a parameter to TAL_PROC   */
}
```

When you pass a C routine as a parameter, the compiler passes a 32-bit address that contains PEP and map information in the high-order word and a zero in the low-order word.

Differences Between Native and TNS Mixed-Language Programs

The information in this section is for writing mixed-language native C or C++ programs. If you need to write mixed-language native C or C++ programs that can run as both TNS and native processes, you must also read **Mixed-Language Programming for TNS Programs** on page 113.

To write mixed-language programs that run as both TNS and native processes, you must consider at least these issues:

- **Data Models** on page 162
- **Memory Models** on page 162
- **_far Pointer Qualifier** on page 162
- **Extended Data Segments** on page 162

Data Models

The native C and C++ compilers support only one data model: the 32-bit or wide data model. The size of int is always 32 bits. If you are writing C or C++ programs that you want to run as both native and TNS processes, you must write code that expects the size of type `int` to be 32 bits. A good program practice is to use short for all 16-bit integers and int for all 32-bit integers.

Memory Models

The native C and C++ compilers support the large-memory model.

The TNS C compiler supports the small-memory model or the large-memory model, depending on the amount of data storage required. However, the large-memory model is recommended and is the default setting.

The size of a pointer in the large-memory model is 32 bits. The size of a pointer in the small-memory model is 16 bits.

If you are writing native C or C++ programs that you want to run as both native and TNS processes, you need to write programs that use the large-memory model.

_far Pointer Qualifier

For native C and C++, there is no need to specify the `_far` pointer qualifier because the native compilers support only the large-memory model.

For TNS C, you must use the _far pointer qualifier in the parameter-type-list of an interface declaration to specify a parameter type that is defined in TAL as an extended pointer using .EXT. For example:

```
_tal _variable _cc_status PROC_3 (short _far *);
```

If you are writing native C or C++ programs that you want to run as both native and TNS processes, you need to include the _far pointer qualifiers. The native C and C++ compilers accept the _far specifier for compatibility with the TNS compilers.

Extended Data Segments

When the user data segment (TNS processes) or globals area (native processes) does not provide enough data space for your process, you can make additional virtual memory available to the process. Virtual memory is allocated as one or more extended data segments. There are two types of extended data segments: flat segments and selectable segments. Selectable segments are a carryover from TNS system architecture. They continue to be supported on native systems. However, programs written for native systems should use flat segments.

The term “extended” has little significance in the context of native systems. The only non-extended data segment in user address space is the user data segment of a TNS process, and there are no non-extended data segments in a native process.

For more details on using extended data segments, see the “Managing Memory” section in the Guardian Programmer’s Guide.

Interfacing to Native COBOL

Your native C/C++ programs can call functions written in native COBOL.

The general procedure consists of these steps:

- Use a native COBOL compiler to compile the COBOL function.
- Use a native C/C++ compiler to compile the C/C++ program.
- Use `xld`, `eld`, `ld`, or `nld` to link the object files and create the executable.

C Program That Calls a COBOL Function on page 163, **COBOL Function Called by a C Program** on page 163, and **Include File (Prototype Function)** on page 164 shows the contents of three files: a C program, a COBOL program, and a header file:

File Type	OSS Name	Guardian Name
C program	`testc.c`	`testc`
COBOL program	`testcob.cob`	`testcob`
Header or include file containing prototype function	`cobincl.h`	`cobincl`

In the example files shown on the following pages, file names appear as the OSS versions. If you are working in the Guardian environment, the file names would be the Guardian versions.

For examples of using TNS C programs to call TNS COBOL functions, see **Interfacing to TNS COBOL** on page 138.

C Program That Calls a COBOL Function

```
/* testc.c */

/* #pragma inspect, symbols */
/* #pragma search largec    */
#include <stdio.h>
#include "cobincl.h"

short main (void)
{

  short ds;
  long dl;
  char *tx = "Displayed in COBOL";
  ds = 100;
  dl = 40000;
  XCOBFUNC(tx, &ds, &dl);

printf("I am back in C now and program is ending.\n");
}
```

COBOL Function Called by a C Program

```
The COBOL program for OSS. testcob.cob
================================================================
?env common;innerlist
?SYMBOLS
 IDENTIFICATION DIVISION.
 PROGRAM-ID. XCOBFUNC.
 AUTHOR. MOLLY.
 DATE-WRITTEN. 7/25/00.
```

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
  SOURCE-COMPUTER. TANDEM-K2006.
  OBJECT-COMPUTER. TANDEM-K2006.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 D-RESULT              PIC S9(09) COMP.

LINKAGE SECTION.
77 D-STRING              PIC X(20).
77 D-SHORT               NATIVE-2.
77 D-LARGE               NATIVE-4.

PROCEDURE DIVISION USING D-STRING, D-SHORT, D-LARGE.
000-INIT.
    DISPLAY "I AM DOING COBOL NOW".
    COMPUTE D-RESULT = D-LARGE / D-SHORT.
    DISPLAY "D-STRING = " D-STRING.
    DISPLAY "D-LARGE  = " D-LARGE.
    DISPLAY "D-SHORT  = " D-SHORT.
    DISPLAY "D-RESULT = " D-RESULT.
    DISPLAY "LEAVING COBOL PROGRAM NOW".
    EXIT-PROGRAM.
```

Include File (Prototype Function)

```
================================================================
The include file for OSS (cobincl.h)
================================================================
/* COBINCL */
void XCOBFUNC (char *, short *, long *);
#pragma FUNCTION XCOBFUNC (cobol)
```

Compiling and Linking the COBOL and C Programs

To compile the programs in the OSS environment:

1. Compile the COBOL function named `testcob.cob` using the TNS/R native COBOL compiler:

   ```
   nmcobol -c testcob.cob
   ```

   or compile the function using the TNS/E native COBOL compiler:

   ```
   ecobol -c testcob.cob
   ```

   or compile the function using the TNS/X native COBOL compiler:

   ```
   xcobol -c testcob.cob
   ```

2. Compile the C program named `testc.c` using `c89` or `c99` (TNS/E or TNS/X only):

   ```
   c89 -c testc.c
   ```

   The resulting object file is named `testc.o`.

3. Link the object files using the TNS/R native linker on a TNS/R system or `eld` on a TNS/E system or `xld` on a TNS/X system :

   ```
   nld /usr/lib/crtlmain.o testc.o testcob.o -o test.exe -obey /
   G/system/system/libcobey -l zcobsrl
   ```

   or

   ```
   eld /usr/lib/ccplmain.o testc.o testcob.o -o test.exe
   -l zcobdll
   ```

   or

   ```
   xld /usr/lib/ccpmainx.o testc.o testcob.o -o test.exe
   -l xcobdll
   ```

4. Run the program:

   ```
   ./test.exe
   ```

To compile the programs in the Guardian environment:

1. Compile the COBOL function named `testcob` using the TNS/R native COBOL compiler:

   ```
   nmcobol /in testcob / testcobo
   ```

   or compile the function using the TNS/E native COBOL compiler:

   ```
   ecobol /in testcob / testcobo
   ```

   or compile the function using the TNS/X native COBOL compiler:

   ```
   xcobol /in testcob / testcobo
   ```

2. Compile the C program named `testc` using the TNS/R native C compiler:

   ```
   nmc / in testc/ testco
   ```

   or compile the function using the TNS/E or TNS/X native C compiler:

   ```
   ccomp / in testc/ testco
   ```

3. Link the object files using TNS/R native linker on a TNS/R system or `eld` on a TNS/E system or `xld` on a TNS/X system :

   ```
   nld $system.system.crtlmain testco testcobo -o testexe -obey
   $system.system.libcobey -l zcobsrl
   ```

   or

   ```
   eld $system.system.ccplmain testco testcobo -o testexe
   -l zcobdll
   ```

   or

   ```
   xld $system.system.ccpmainx testco testcobo -o testexe
   -l xcobdll
   ```

4. Run the program:

   ```
   run testexe
   ```

System-Level Programming

System-level programming refers to the ability to write TNS C and C++ functions that reside in system code, system library, or user library. It also refers to native C and C++ functions that reside in user library. Customer-written native C and C++ functions cannot reside in system code or system library. System-level programming is available only in the Guardian environment.

Specifying a Code Space

A code space is a part of virtual memory that is reserved for user code, user library code, system code, and system library code. Depending upon the code space, the C compiler generates different code and places different restrictions on your code. Use the `ENV` pragma to specify the code space.

To compile functions to run in system code, specify `ENV EMBEDDED`. To compile functions to run in system library, specify `ENV LIBSPACE`. To compile functions to run in user library, specify `ENV LIBRARY`. For more details, see the description of the pragma **ENV** on page 221.

For each code space, there are restrictions on the run-time library and language features that can use be used. **Code Spaces and the Availability of Run-Time Library and Language Features** table summarizes the availability of run-time library and language features for each code space.

Table 23: Code Spaces and the Availability of Run-Time Library and Language Features

Feature	System Code	User Library	System Library
C I/O functions	No	Yes *	No
Memory allocation functions	No	Yes	No
Main routine	No	No	No
Relocatable data blocks	Yes	No	No
Functions that set `errno`	No	Yes	No
CRE library functions	No	Yes	No
Constants allocated in code space	No **	Yes	Yes
Constants allocated in data space	Yes	No	No

* TNS C and C++ programs can use functions in a user library if direct access to relocatable data blocks is not needed for the operations. Native C and C++ programs in a user library can use any C library function.

**The `_cspace` type qualifier lets you allocate constants in the code space.

Because of these restrictions on resources, functions that run in user library, system library, or system code cannot use the complete C run-time library. For example, functions that are dependent upon their execution context because their semantics require references to external data can run only in user code or system code.

Passing Pointers to Constants Between Code Spaces

A code space is a part of virtual memory that is reserved for user code, user library code, system code, and system library code. A data space is an part of virtual memory that is reserved for user data and system data. For functions that reside in the user code space, the compiler places variables of type `const`, constant values on the right-hand side of initialization statements, and strings in the user data space.

For functions that reside in the user library space or system library space, the compiler places variables of type `const`, constant values on the right-hand side of initialization statements, and strings in the user code space. (User library and system library do not allow relocatable data blocks. Without relocatable data blocks, static data is not allowed. Therefore, the compiler allocates static data items such as constants in the user code space instead of the user or system data spaces.)

To pass pointers to such constants in TNS C and C++ programs, you must use the `CSADDR` pragma. (Native C programs do not require the `CSADDR` pragma.) The `CSADDR` pragma directs the compiler to allocate sufficient space on the stack for each code space data object pointed to by a function argument. It generates move instructions to copy each data object to the corresponding implicit stack address, based on information from previous data declarations. The new stack address, instead of the code space address, is then passed as the function argument.

Example

This example shows valid uses of the `CSADDR` pragma to pass pointers to constants between code spaces:

```
#pragma env library
#pragma csaddr
#pragma xmem
void foo( const void * );

main()
{
   const struct S { int i,j,k; } s = { 10, 20, 30 };
   const struct S *sp;

   foo( "This is a test" ); /* 15 bytes copied and passed */

   foo( &s );   /* 6 bytes copied and passed */

   sp = &s;
```

A pointer variable to the code space cannot be initialized. Explicit assignment or conversion to `array` type is recommended. For example, this function results in an error because `name` is an array of pointers and therefore cannot be initialized:

```
#pragma env library
#include <stringh>
int find_kevin(void)
{
   const char *name[] =
      {"linda", "don", "kevin"};
   const int LAST_NAME = 3;
   const char *p;
   int i = 0;

   for (p = name; i <= LAST_NAME ; i++, p = &name[i])
      if (!strcmp(p, "kevin"))
         return TRUE;
      return FALSE;
}
```

The previous function can be made valid by converting `name` to a two dimensional array without making any other changes, as shown in this code:

```
#pragma env library
#pragma inline
#include <stringh>
int find_kevin(void)
{
   const char name[][6] =
      {{"linda"}, {"don"}, {"kevin"}};
   const int LAST_NAME = 3;
   const char *p;
   int i = 0;

   for (p = name; i <= LAST_NAME ; i++, p = &name[i])
      if (!strcmp(p, "kevin"))
         return TRUE;
      return FALSE;
}
```

Note the inclusion of pragma inline in the preceding compilation unit. Because the passing of pointers to code space is hazardous for the run-time function call of `strcmp`, generation of inline code is safer.

You can also place constants in the code space with the `_cspace` type qualifier. However, the `_cspace` type qualifier is effective only on the specified data constant. Other constants in the function without the type qualifier are still allocated in the static data space. The use of the _cspace qualifier under the `ENV LIBRARY` or `ENV LIBSPACE` pragmas is redundant for object definitions with the type qualifier `const`, except for pointers. For example, you must specify `_cspace` in this declaration to store the object in the code space, regardless of specified `ENV` pragma:

```
#pragma env libspace
const char * lvar = "x";           /* Stored in data space */
const _cspace char * hvar = "y";   /* Stored in code space */
```

1. This example declares the `arr` constant array to be located in the current code space:

   ```
   _cspace const char arr[] = "ABCD";
   ```

2. This example declares a pointer data type to point to a constant item located in the code space:

   ```
   extern _cspace const int * ptr;
   ```

If you apply the & operator to a constant stored in the code space, the result is a 32-bit extended pointer.

Writing Variable and Extensible Functions

Variable and extensible functions are HPE extensions to the ISO/ANSI C language standard. This subsection describes how to use variable and extensible functions, including:

- Declaring variable functions with the `variable` attribute
- Declaring extensible functions with the `extensible` attribute
- Checking for actual parameters with the `_arg_present()` operator
- Omitting parameters on calls to variable and extensible functions
- Converting variable functions to extensible functions

For a complete syntax description for writing variable and extensible function declarations, see **FUNCTION** on page 239.

For native C, variable and extensible functions are treated the same. The semantics used are those for extensible functions. Therefore, declare procedures extensible instead of variable in native C programs. This discussion about variable and extensible functions applies only to TNS C and C++, not to native mode.

Declaring Variable Functions

The `variable` attribute directs the compiler to treat all parameters of a function as though they are optional, even if some are required by your code. If you add parameters to a variable function declaration, all procedures that call it must be recompiled. This example declares a variable function:

```
foo(int i, int j, int k);
#pragma function foo (variable)
{ ...  /* foo declaration */ }
```

Specify the `FUNCTION` pragma at function declaration, not at function definition.

When you call a variable function, the compiler allocates space in the parameter area for all the parameters. The compiler also generates a parameter mask, which indicates which parameters are actually passed. You use the `_arg_present()` built-in operator to test for the presence of actual parameters.

Declaring Extensible Functions

The `extensible` attribute enables you to add new parameters to a function declaration without recompiling its callers. The compiler treats all parameters of a function as though they are optional, even if some are required by your code. This example declares an extensible function:

```
foo(int i, int j, int k);
#pragma function foo (extensible)
{ ...  /* foo declaration */ }
```

Specify the `FUNCTION` pragma at function declaration, not at function definition.

When you call an extensible function, the compiler allocates space in the parameter area for all the parameters. The compiler also generates a parameter mask, which indicates which parameters are actually passed. You use the `_arg_present()` built-in operator to test for the presence of actual parameters.

Checking for Actual Parameters With _arg_present()

Each variable and extensible function must check for the presence or absence of actual parameters that are required by the function's code. The function can use the `_arg_present()` operator to check for required or optional parameters. The operator `_arg_present()` returns a zero if the parameter is absent; it returns a nonzero `int` value if the parameter is present. The operand to `_arg_present()` must be a valid formal parameter. The `_arg_present()` operator is a built-in operator; a header file is not necessary.

In this example of a variable function, a default value is assigned if the first argument is absent. It operates on the second argument if it is present.

```
void errmsg (int x, int p1, int p2);
#pragma function errmsg (variable)
         /* function declaration */

_variable void errmsg (int x, int p1, int *p2)
 /* function definition */

#define P1_DEFAULT_VALUE -1
{
   if (!_arg_present(p1))      /* Valid, use default value */
   {  p1 = P1_DEFAULT_VALUE;   /* if p1 is absent */
      ...
   }
   if (_arg_present(p2))     /* Valid, use the passed value */
   {  *p2 = 10;              /* if p2 is present */
      ...
   }
}
```

Omitting Parameters

When you call a variable or extensible function, you can omit parameters indicated as being optional in the called function. To omit parameters, use a place-holding comma for each omitted parameter up to the last specified parameter. Here is an example:

```
void someproc (int x, int length, int limit, int total);
#pragma function someproc (extensible)
{ ...  /* function declaration */ }

{
   /*  Lots of code  */
}

void anotherproc (int i);
{  int total;
   total = 259;
   /*  Some code  */
   someproc (23,,,total);
   /*  Omit length and limit parameters  */
}
```

Converting Variable Functions to Extensible Functions

You can convert a TNS C or C++ variable function into an extensible function. When you do so, the compiler converts the `_variable` parameter mask into an `_extensible` parameter mask. Converting a native C or C++ variable function into an extensible function is unnecessary, because native C and C++ treat functions declared variable or extensible as extensible functions.

NOTE: To convert a variable function into an extensible function, you must use the deprecated syntax for writing function declarations. This syntax uses the `_variable` and `_extensible` attributes described in **Attribute Specifier** on page 61. You cannot use the `FUNCTION` pragma in this case. Isolate use of this deprecated declaration syntax.

Converting a variable function to extensible is the only way to add parameters to the function without recompiling all its callers. You can add new parameters at the time you convert the function or later.

You can convert any variable function that meets these criteria:

- It has at least one parameter.
- It has no more than 16 words of parameters.
- All parameters are one word long except the last, which can be one word or longer.

The size of a formal reference parameter is one word for the small-memory model and two words for the large-memory model.

To convert an existing variable function to extensible, redeclare the function and add these information:

- Any new formal parameters
- The keyword `_extensible`
- The number of formal parameters in the variable function, specified as an `int` value in the range 1 through 15 and enclosed in parentheses

This example converts a variable function to an extensible function and adds a new formal parameter, `p3`. The value 2 in parentheses specifies that the function had two formal parameters before it was converted.

```
_c _extensible (2) void errmsg (int p1, char *p2, int p3);
{
   /*  Lots of code  */
}
```

Converting C-Series TNS Programs to Use the Current TNS Compiler

This chapter lists the C language-specific changes that you must make to C-series TNS programs to compile and run them as TNS programs on current systems. Most of these changes were made to TNS C to make it compliant with the ISO/ANSI C standard. For a description of Guardian-specific changes, see the *Guardian Application Conversion Guide.*

NOTE: Do not convert a C-series TNS C program to a current native C program directly. First convert the program from C-series TNS C to current TNS C and then to native C.

- Replace references to the `LARGEC`, `SMALLC`, and `WIDEC` memory-model library files with the `CLARGE`, `CSMALL`, and `CWIDE` files.
- Replace the `min` and `max` macros with the `_min` and `_max` macros. The behavior of these macros has not changed.
- Make these keyword changes:
  - Replace `cc_status` with `_cc_status`.
  - Replace `lowmem` with `_lowmem`.
  - Replace `extensible` with `_extensible`.
  - Replace `variable` with `_variable`.
  - Replace `tal` with `_tal`.

    The current TNS compiler issues a warning message if it finds an old keyword.

- Change code that relies on negative integral values being assigned to `errno`.

  The ISO/ANSI C standard specifies that only positive integral values can be assigned to `errno`. To conform to this requirement, changes were made in the `errnoh` header file. If your C program source explicitly uses a literal defined in `errnoh`, the changes cause the code in the object file to be incompatible with the run-time library. For example:

  ```
  if (errno == ERANGE)  {...}
  ```

  compiles into code dependent upon the value denoted by `ERANGE`. `ERANGE` has changed, so you must recompile this program. However, you need not recompile a program that checks only that the value of `errno` is 0 or not equal to 0.

- Change code that relies on the type of `size_t`, the result of the `sizeof` operator. The type `size_t` has changed from type *long* to type `unsigned long`. This change applies to the large-memory model (pragma `XMEM`) and the 32-bit or wide data model (pragma `WIDE`).
- Change code that uses the result of the `sizeof` operator for constant operands under the 32-bit or wide data model (pragma `WIDE`):
  - For C-series compilers, the `sizeof` operator returns 2 or 4 for constant operands.
  - For the current compilers, the `sizeof` operator returns 4 for all constant operands.

- Change the use of type `long` as a bitfield type under the 32-bit or wide data model to type `int`. The ISO/ANSI C standard supports only type `int` as a bitfield type. However, the current TNS compiler continues to support bitfields of type `short` for the large-memory model.
- Specify a function prototype within the scope of each call to a function.

  The C-series compiler generates an implicit function prototype at the call to a function that does not have a prototype specified in the scope of the call. This implicit prototyping violates the ISO/ANSI C standard and results in code that is not portable. The current TNS compiler does not generate these function prototypes for you.
- Change code that is affected by the behavioral changes of the `sscanf()`, `fflush()`, or `fcntl()` functions:
  - For C-series compilers, `sscanf()` returns a -1 if no conversion takes place. For the current compilers, `sscanf()` returns a 0 if no conversion takes place.
  - For C-series compilers, the *fflush()* function can flush partial lines if standard output (`stdout)` and standard error (`stderr`) are directed to type 101 EDIT files. For the current compilers, the `fflush()` function does not flush partial lines if `stdout` and `stderr` are directed to type 101 EDIT files because these files are shared resources under the CRE.

    For the current compilers, to flush partial lines to the `stdout` and `stderr` files, you must either open the files using `freopen()` or close and reopen the files using `fopen()`. In both cases, these files cannot be shared.
  - For C-series compilers, the `fcntl()` function with the `F_GETDTYPE` parameter returns the device type of the file in bits 4 through 9 of the value returned. It also returns additional information from the Guardian procedure DEVICEINFO in other bits of the returned value.

    For the current compilers, the `fcntl()` function returns only the device type information in the value returned. Instead of calling Guardian procedure DEVICEINFO, `fcntl()` calls the Guardian procedure FILE_GETINFO_.
- Change macros to use a `#` directive to replace parameters inside a literal string.

  For C-series compilers, if a function-like macro has a literal string in its substitution list, the compiler treats the literal string as a series of tokens. The compiler searches the literal string for a token matching the macro parameter list. If the compiler finds a match, it performs the substitution.

  For the current compilers, the compiler treats a literal string as a preprocessor token of the substitution list. The literal string is not scanned.

  For example, you must change this C-series macro to use the `#` directive:

  ```
  C-series macro:
  #define pr (x, format) printf("The x = %format\n"), (x))

  D-series macro:
  #define str(a) # a
  #define pr(x, format) printf(str(The x = %format\n), (x))
  ```
- Recode your programs to eliminate the use of these HPE C supplementary library calls that are not supported by the current compilers:

  - ◦ extfname_to_intfname() This function is not required because the D-series Guardian system procedures accept and return external file names.
  - ◦ lastreceive() and receiveinfo() Call the Guardian procedure FILE_GETRECEIVEINFO_ instead.

- Before the D-series releases, you had to bind the memory-model independent C run-time library (CLIB) into your TNS application code. On current systems, CLIB and the Common Run-Time Environment library (CRELIB) are configured as part of the system library. (These run-time library routines have already been accelerated for native systems.) Do not bind CLIB or CRELIB into current programs. Check any Binder scripts for C-series compilations and remove any of these statements:

  ```
  SELECT SEARCH clib
  ADD * FROM clib
  ```

These additional changes were made to HPE C for the D-series releases. It is highly unlikely that you must change your programs because of these changes:

- The definition of the object-like macro NULL was removed from the C header files asserth, ctypeh, errnoh, floath, limitsh, mathh, and setjmph. These header files now conform to the ISO/ANSI C standard.
- The number of parameters allowed for a nonextensible and nonvariable functions increased from 200 to 252. The compiler issues an error if a function exceeds this limit.
- The default number of secondary extents for files created by the Guardian C fopen() and creat() functions increased from 4 to 16.
- The default number of maxextents for files created by the Guardian C fopen() and creat() functions increased from 16 to 500.
- Starting with the D20.00 RVU, the C run-time library uses Greenwich Mean Time (GMT) for the date January 1, 1970. In previous releases, the C run-time library used Pacific standard time (PST), which is eight hours ahead of GMT. The ISO/ANSI C standard does not specify GMT. However, virtually all software vendors use GMT, therefore, the HPE implementation has changed. Greenwich Mean Time is also called Coordinated Universal Time (UTC).

Migrating Programs to TNS/R, TNS/E, or TNS/X Native Mode

HPE NonStop native mode enables you to write programs that are fully optimized for TNS/R, TNS/E, or TNS/X systems such as NonStop servers. The term TNS/R native means the program uses the process, memory, and instruction set architectures that are native to RISC processors. The term TNS/E native means the program uses the process, memory, and instruction set architectures that are native to Intel Itanium processors. The term TNS/X native means the program uses the process, memory, and instruction set architectures that are native to Intel x86 processors.

Native compilers and tools are used to generate native programs. Other tools have been enhanced to support native programs. Native programs can be written in the native Portable Transaction Application Language, pTAL (a dialect of the Transaction Application Language, TAL), C, C++, and native COBOL.

Common tasks required to migrate a TNS C or C++ program to native mode include:

- Convert programs to use 32-bit pointers and values of type `int` (the wide or 32-bit data model and the large-memory model).
- Replace obsolete keywords and external function declarations.
- Change code that uses `_cc_status` for return values.
- Change code that relies on arithmetic overflow traps.
- Replace calls to C library functions that cannot be called by native programs.
- Remove obsolete pragmas.
- Specify most pragmas on the command line instead of in source files.

The TNS and native C compilers conform to the ISO/ANSI C language standard. Source code that compiles without warnings or errors with the TNS C compiler or C++ preprocessor might get warnings or errors using the native C and C++ compilers. (In most cases, the native compilers are better than the TNS compilers in detecting violations to the ISO/ANSI C standard.)

The native mode C and C++ migration tool, NMCMT, can help you migrate programs to native mode. The migration tool scans source files and produces a diagnostic listing. The diagnostic listing identifies C and C++ language source code changes required to migrate from TNS C and C++ to native C and C++.

The migration tool detects most but not all changes required to migrate a TNS program to a native program. The migration tool does not detect changes required in areas outside the C and C++ languages, such as the Guardian and CRE application program interface (API), or related to converting programs to use the 32-bit data model. For conversion details, see **Converting Programs to the ILP32 Data Model** on page 435.

You can use the migration tool in either the Guardian or the OSS environment.

For more detailed information on:

Running the migration tool and on migrating TNS programs to TNS/R native mode, see the *TNS/R Native Application Migration Guide*.

Migrating TNS/R programs to TNS/E native mode

See the *H-series Application Migration Guide*.

Migrating TNS/E programs to TNS/X native mode

See the *L-series Application Migration Guide*.

Preprocessor Directives and Macros

The preprocessor is a macro processor that processes the source text of a program before the compiler parses the program.

The preprocessor is controlled by preprocessor directive lines that appear in your source text. Preprocessor directive lines always begin with a number sign (#) in column one. The name of the directive follows the number sign. The remainder of the line can contain arguments for the directive.

Preprocessor Directives table summarizes the preprocessor directives.

Table 24: Preprocessor Directives

`#define`	Defines a preprocessor macro and defines an identifier as the macro name.
`#elif`	Conditionally includes text, depending on the value of a constant expression.
`#else`	Conditionally includes text if the test on the related `#if, #elif, #ifdef,` or `#ifndef` directive fails.
`#endif`	Terminates conditional text.
`#error`	Forces a compilation error and terminates compilation.
`#if`	Conditionally includes text, depending on the value of a constant expression.
`#ifdef`	Conditionally includes text, depending on whether an identifier is currently defined as a macro name.
`#ifndef`	Conditionally includes text, depending on whether an identifier is currently undefined as a macro name.
`#include`	Replaces the directive with the contents of a specified file.
`#line`	Causes the compiler to renumber the lines in the source text.
`#pragma`	Introduces a compiler pragma.
`#undef`	Deletes a macro definition.

#define

The `#define` directive defines a macro, providing it with a name (an identifier) and the replacement list that the name represents. The macro can be either object-like or function-like, depending on how you define it.

```
#define { object-like-name   } replacement-list newline
        { function-like-name }

object-like-name:
   identifier
function-like-name:
   identifier( [parameter-list] )
```

object-like-name

specifies the name of an object-like macro; it must be a valid identifier.

function-like-name

specifies the name and parameters (if any) of a function-like macro. The parameter list is a parenthesized, comma-separated list of zero or more identifiers. Note that there can be no white space between the macro identifier and the opening parenthesis of the parameter list.

replacement-list

is the text that the preprocessor substitutes for the macro name when you invoke the macro later in the translation unit. The *replacement-list* must be valid in the context where you invoke it, not where you define it.

If the macro is function-like and has defined parameters, the preprocessor substitutes each instance of a parameter identifier in the *replacement-list* with its corresponding argument specified in the macro invocation. The second example illustrates parameter substitution.

newline

is the newline character that terminates the directive line.

Usage Guidelines

- If you need more than one physical line to complete the definition of a macro, place a backslash (\) at the end of all but the last line of the definition. The backslashes cause these physical lines to be concatenated into a single logical line.
- The `#ifdef`and `#ifndef` directives enable you to test whether an identifier is currently defined as a macro name.
- The `#undef` directive enables you to remove a macro definition so that the identifier is no longer defined as a macro name.
- Unless you delete a macro definition, it remains in effect for the remainder of the translation unit.
- Avoid invoking macros with arguments that cause side effects. When the macro evaluates an argument more than once, it also produces the side effect more than once, sometimes with unexpected results. For example, if you invoke the `sqr` macro (defined in the second example) with argument `x++`:

  ```
  y = sqr(x++);
  ```

  the preprocessor expands it to:

  ```
  y = ((x++) * (x++));
  ```

  This expansion causes the side effect (increment of x) to occur twice.

Examples

1. This example defines the object-like macro EOF to represent the constant `-1`:

   ```
   #define EOF (-1)
   ```

2. This example defines the function-like macro `sqr` and then uses it in an assignment:

   ```
   #define sqr(parm) ((parm) * (parm))

   int main(void)
   ```

```
{
   int a,b;

   a = 12;
   b = sqr(a);
}
```

After expanding the macro invocation and substituting the argument for the parameter in the replacement list, the assignment to b looks like this:

```
b = ((a) * (a));
```

#error

The #error directive allows you to force a compilation error and terminate compilation.

```
#error preprocessor-tokens
```

preprocessor-tokens

specifies the tokens to be included in the message text.

Example

This example causes the string following #error to be printed as the error message:

```
#error This message will be issued by the compiler
```

#if, #elif, #ifdef, #ifndef, #else, and #endif

The four if directives, together with the #else and #endif directives, define the bounds of an if section, which conditionally includes or excludes source text. An if section consists of:

- An if group, which is one of the four if directives followed by the source text that the compiler is to include if the controlling condition is true
- An optional else group, which is the #else directive followed by the source text that the compiler is to include if the controlling condition is false
- The #endif directive, which marks the end of the if section

The four if directives offer different controlling conditions:

#if	Tests the value of a constant expression. If the value is nonzero, the source text in the if group is included; otherwise, the source text in an elif or else group, if present, is included.
#elif	Tests the value of a constant expression. If the value is nonzero, the source text in the elif group is included; otherwise, the source text in the next elif or else group, if present, is included.

Table Continued

`#ifdef`	Tests the existence of an identifier as a macro name. If the identifier has been defined (using `#define`), the source text in the if group is included; otherwise, the source text in the else group (if present) is included.
`#ifndef`	Tests the nonexistence of an identifier as a macro name. If the identifier has not been defined, the source text in the if group is included; otherwise, the source text in the else group (if present) is included.

```
if-section:
if-group
[ else-group ]
 #endif newline

if-group:


  { #if int-constant-expression   } newline [ source-text ]
  { #elif int-constant-expression }
  { #ifdef identifier             }
  { #ifndef identifier            }

#else-group:

 #else newline [ source-text ]
```

`#if` ***int-constant-expression newline [source-text]***

introduces an if section that conditionally includes source text based on the value of a constant expression. The new line following the constant expression terminates the `#if` directive line, and *source-text* is the text included if *int-constant-expression* has a nonzero value.

The constant expression must evaluate to an integral value; also, it cannot include `sizeof` or cast operators or an enumeration constant.

`#elif` ***int-constant-expression newline [source-text]***

introduces an if section that conditionally includes source text based on the value of a constant expression. The new line following the constant expression terminates the `#elif` directive line, and *source-text* is the text included if *int-constant-expression* has a nonzero value.

The constant expression must evaluate to an integral value; also, it cannot include `sizeof` or cast operators or an enumeration constant.

The maximum number of nested `#elif` directives is 32.

`#ifdef` ***identifier newline [source-text]***

introduces an if section that conditionally includes source text based on the existence of an identifier as a macro name. *identifier* specifies the identifier to test, and the new line following it terminates the `#ifdef` directive line. *source-text* is the source text that is included if *identifier* is currently defined as a macro name.

`#ifndef` ***identifier newline [source-text]***

introduces an if section that conditionally includes source text based on the nonexistence of an identifier as a macro name. *identifier* specifies the identifier to test, and the new line following it terminates the `#ifndef` directive line. *source-text* is the source text that is included if *identifier* is not currently defined as a macro name.

#else *newline [source-text]*

is the else group of an if section. The new line following the #else directive terminates the directive line. *source-text* specifies the source text that is included if the condition specified in the if group is not met.

#endif *newline*

terminates the if section. The new line following the #endif directive terminates the directive line.

Usage Guidelines

- When using the if directives, remember to distinguish between macro definitions and function definitions; the #ifdef and #ifndef directives test only macro definitions.
- The preprocessor selects a single *source-text* evaluating the constant expression following each #if or #elif directive until it finds a true, nonzero, *constant-expression*. It selects all text up to its associated #elif, #else, or #endif.
- If all occurrences of *constant-expression* are false, or if no #elif directives appear, the preprocessor selects the source text after the #else clause. If the #else clause is omitted and all instances of *constant-expression* in the #if blocks are false, no source text is selected.

Examples

1. This example shows how the #if, #else, and #endif directives interact. Because the identifier ANSI is defined as zero (false), the #if test fails. As a result, the compiler processes only the source text in the else group:

```
#define ANSI 0

#if ANSI
  printf("Function prototypes supported.\n");
#else
  printf("Function prototypes not supported.\n");
#endif
```

2. This example shows how the #elif directive works. The #if, #elif, and #else directives are used to make one of three choices, based on the value of XCOORD and YCOORD. Note that XCOORD and YCOORD must be defined constants.

```
#define XCOORD 5
#define YCOORD 5

#if XCOORD == 10
    printf("Intersection at (10,5).\n");
#elif YCOORD == 10
    printf("Intersection at (5,10).\n");
#else
    printf("Intersection at (5,5) \n");
#endif
```

3. This example shows how the #ifdef directive works. Because PC is defined as a macro, the compiler processes the source text following the #ifdef PC directive. The compiler ignores the source text following the #ifdef TNS directive because TNS is undefined:

```
#define PC 1
```

```
#ifdef TNS
   /* HP Tandem dependent code. */
   printf("This program executes on the server.\n");
#endif

#ifdef PC
   /* pc dependent code. */
   printf("This program executes on the pc.\n");
#endif
```

4. This example shows how the `#ifndef` directive works. Because `TNS` is not defined as a macro, the compiler processes the source text following the `#ifndef TNS` directive. The compiler ignores the source text following the `#ifndef PC` directive because `PC` is defined:

```
#define PC 1

#ifndef TNS
   printf("TNS is not currently defined.\n");
#endif

#ifndef PC
   printf("PC is not currently defined.\n");
#endif
```

#include

The `#include` directive includes source text from a specified file.

```
#include source_specifier [ nolist ] newline

source-specifier:
   { " source_file [ section_list ] "         } |
   { < library_header_file [ section_list ] > }

section_list:

( section_name [ , section_name ]... )
```

source_specifier

specifies the location of the program text that the compiler includes in the compilation.

source_file

is the file name of the source file you want to include. In the Guardian environment, the compiler searches for *source_file* using the **SSV** on page 308 search list. In the OSS environment, the compiler searches for *source_file* using operands specified in the `c89` or `c99` utility `-I` flag.

library_header_file

is the name of the library header file you want to include. The compiler assumes that library header files reside in the same volume and subvolume (in the Guardian environment) or directory (in the OSS environment) as the compiler. In the Guardian environment, the compiler searches for *library_header_file* using the **SSV** on page 308 search list. In the OSS environment, the compiler searches for *library_header_file* using operands specified in the `c89` or `c99` utility `-I` flag.

section_list

is a parenthesized, comma-separated list of section names found in the specified source file or library header file. Sections are created using the `SECTION` pragma, which is discussed in **SECTION** on page 299.

This is an HPE NonStop extension to the standard.

`nolist`

directs the compiler not to list the contents of the file or sections being included.

This is an HPE NonStop extension to the standard.

newline

is the newline character that terminates the directive line.

Usage Guidelines

- You can enter the `#include` directive on the command line or in the source text.
- The `#include` directive specifies a pathname that the compiler searches before looking in the usual places.
- The way that you specify a # `include` directive affects the search:
  - For the TNS C compiler, the `include` directive searches for the specified file (in double quotes) in the current default Guardian volume and subvolume. If the file is not found, the directive searches in the compiler's Guardian volume and subvolume (e.g. `$system.system`). If pragma SSV< *n* > is used, the Guardian subvolumes are then searched.
  - For the TNS/R C compiler, the TNS/E C compiler, and the TNS/X C compiler, the `include` directive searches for the specified file (in double quotes) in the Guardian volume and subvolume or in the OSS or Windows directory containing the source file.

    If the file is not found, the directive searches in the compiler's Guardian volume and subvolume (e.g. `$system.system`) or in the OSS or Windows directory (e.g. `$COMP_ROOT/usr/include` , the OSS or Windows default location, with `$COMP_ROOT` being a user-defined environment variable).

    NOTE:

    – On OSS, standard headers are installed in the OSS default location. Typically on OSS systems, this environment variable is not explicitly defined, so the default location gets expanded to `/usr/include.`

    – On Windows, standard headers are not installed in `/usr/include`. So, if you do not define the Windows environment variable, you will need to explicitly add the location of the standard headers using the `-I` directive when you invoke the compiler.

    If pragma SSV< *n* > (for Guardian) or the `c89` or `c99` `-I` flag (for OSS or Windows) is used, the Guardian subvolumes or the OSS or Windows directories are then searched.
- The compiler searches for files:

File	File Search
`#include` <*library_header_file*>	The specified standard header file is searched for in the location of the compiler (see details in following bullets).
`#include "`*source_file*`"`	The specified user-defined file is searched for in the current default Guardian volume and subvolume or OSS working directory.
`#include "`*subvolume.file*`"`	The specified user-defined file is searched for in the current default Guardian volume.

In RVUs preceding D30.00, if the `#include` specification was of the form `#include "`*subvolume.file*`"`, the Guardian compiler checked for the subvolume in the current volume by default. Beginning with D30.00, you must specify the default volume using an **SSV** on page 308 pragma.

- If you need more than one physical line to complete the section list, place a backslash (\) at the end of all but the last line of the list. The backslashes cause these physical lines to be concatenated into a single logical line. For example, this is translated as a single logical line:

  ```
  #include <cextdecs( \
    PROCESS_GETINFOLIST_,  \
    FILE_OPEN_, \
    WRITE,      \
    WRITEREAD   \
  )>
  ```

- The OSS environment and some operating systems use file names that consist of a name and an extension separated by a period. On such systems, the names of include header files are specified as `name.h`. The C compiler allows this format in the `#include` directive.

  For compilations in the Guardian environment, the C compiler translates `name.h` to *nameh*. For example, `stdio.h` becomes `stdioh`. If the *name* portion exceeds 7 characters, the C compiler truncates it to 7 characters.

- For TNS compilations, included files can be nested to a depth of 16. For example, file `f1` includes file `f2`, file `f2` includes file `f3`, and so on until file `f15` includes file `f16`.
- For TNS compilations, the maximum number of `#include` directives that can appear in a single compilation unit is 2048. If the same include file occurs twice in a compilation unit, that file is counted twice because it appears in two different `#include` directives.

Examples

1. This example includes the `stdioh` header file, specified for the Guardian environment:

   ```
   #include <stdioh>
   ```

2. This example includes the `stdio.h` header file, specified for the OSS environment:

   ```
   #include <stdio.h>
   ```

3. This example includes the file `mysource` from the current volume and subvolume or current working directory:

   ```
   #include "mysource"
   ```

#line

The #line directive causes the compiler to renumber the lines of the source text so that the next line has the specified number and causes the compiler to believe that the current source file name is *file-name*. If *file-name* is not specified, only the renumbering of lines takes place.

```
#line number [file-name]
```

#pragma

The #pragma directive instructs the preprocessor to pass a compiler pragma on to the compiler.

```
#pragma compiler-pragma [ , compiler-pragma ]... newline
```

compiler-pragma

is any compiler pragma described in **Compiler Pragmas** on page 193.

newline

is the newline character that terminates the directive line.

Example

This example enables run-time error checking:

```
#pragma CHECK
```

#undef

The #undef directive deletes a macro definition created using #define.

```
#undef identifier
```

identifier

is the name of the macro to delete.

Usage Guideline

Once you have deleted a macro definition, its identifier no longer exists as a macro name. Consequently, the #ifdef and #ifndef directives will find the identifier to be undefined.

Example

To delete the macro definition with identifier red and print “Red is undefined.”:

```
#define red 1
/* ... */
#undef red

#ifdef red
  printf("Red is defined.\n");
#else
  printf("Red is undefined.\n");
#endif
```

Predefined Macros

The compiler provides six predefined object-like macros. These macros expand to various statistics regarding compilation, as shown in the **Predefined Macros** table. Note that the identifiers for these macros begin and end with two underscores.

Table 25: Predefined Macros

Macro	What It Expands To
`__DATE__`	A `string` literal representing the date of compilation. This string has the form *Mmm dd yyyy*, where the first character of the day (*dd*) is a blank if the day is less than 10.
`__FILE__`	A `string` literal representing the name of the current source file. The file name is qualified up to the volume (or device) name if the file is on the same system as the compiler; otherwise, the file name is qualified up to the system name.
`__FUNCTION__`	A string representing the name of the current function, or an empty string if it appears outside of a function. This macro cannot be redefined.
`__LINE__`	A `long` value representing the current line number in the current source file. If the current source file is an EDIT file, this value is equal to the EDIT line number multiplied by 1000. For any other type of source file, this value is equal to the ordinal value of the line multiplied by 1000.
`__STDC__`	A `long` value signifying that the compiler is to comply with the ISO/ANSI C standard. If the value is 1, the compiler complies with the standard.
`__TIME__`	A `string` literal representing the time of compilation. This string has the form *hh:mm:ss*, where the hour (*hh*) ranges from 0 to 23. The time zone is that of the system on which the compilation takes place.

None of these predefined macros can be removed using `#undef`.

This example demonstrates the use of the __FUNCTION__ macro:

```
#include <stdio.h>
char ch[] = __FUNCTION__;

void foo (void) {
  printf ("Entering function %s\n", __FUNCTION__);
  printf ("ch[] was in function \"%s\"\n", ch);
};
int main (void) {
  foo ();
}
The output from the preceding code is:
Entering function foo
ch[] was in function ""
```

Predefined Symbols

The compiler provides three predefined preprocessor symbols: `__TANDEM`, `__INT32`, and `__XMEM`.

You can use the __TANDEM symbol to increase the portability of your programs. Enclose system-dependent source text in an if section that uses #ifdef or #ifndef to test for the existence of the __TANDEM symbol.

The predefined preprocessor symbol __INT32 is defined by the TNS C compiler and CFront when the WIDE pragma is present. The predefined preprocessor symbol __XMEM is defined by the TNS C compiler and Cfront when the XMEM pragma is present. The native compilers always define the preprocessor symbols __INT32 and __XMEM.

The __INT32 and __XMEM preprocessor symbols affect the visibility of declarations in header files.

Variadic Macros

Two variants of variadic macro definitions can be invoked in native C with a variable number of arguments. To use these extensions, you must compile with extensions enabled.

- One variant of variadic macro is the C9X form, described in section 6.8.3 of the Working Draft, 1997-11-21, WG14/N794 J11/97-158 of the Proposed ISO C Standard.

  For example:

  ```
  #define D(fmt, ...) printf(fmt, __VA_ARGS__)

  /* The "..." matches an arbitrary positive number of macro
  arguments that can be referred to by __VA_ARGS__
  (includes the separating commas) */

  D("%c%s\n" , ’E’, "DG");
  /* Expands to "printf("%c%s\n", ’E’, "DG");" */

  D("EDG\n");
  /* Expands to "printf("EDG\n", );" -- note the extra comma*/
  ```

- The other variant of variadic macro is the GNU form, provided by the GNU C compiler. This variant adds to the C9X form the ability to name the variadic parameter and a special meaning to the token pasting operator ("##"). When the operator is followed by an empty variadic argument (named or not), the preceding parameter or continuous sequence of nonwhitespace characters (not part of a parameter) is erased.

  For example:

  ```
  #define E(fmt, args...) printf (fmt, ## args)

  E("%c%s\n", ’E’, "DG");
  /* Expands to "printf("%c%s\n", ’E’, "DG");"  */

  E("EDG\n");
  /* Expands to "printf("EDG\n" );" -- no extra comma*/
  ```

Feature-Test Macros

The feature-test macros, shown in the **Predefined Feature-Test Macros** table determine whether a particular set of features will be included from header files. Details about these macros follow this table.

Table 26: Predefined Feature-Test Macros

Macro	What It Defines
`_BOOL`	Defined when bool is a keyword (that is, when using C++ VERSION2 or VERSION3 dialect)
`__CODECOV__`	Defined when the option `CODECOV` is specified.
`__CPLUSPLUS`	Identifiers required to modify C headers for use by C++. (Note double underscore.) This macro is automatically defined by the `c89` or `c99` command when the `-Wcplusplus` flag is used.
`__CPLUSPLUS_VERSION`	Identifier used by the different versions of native C++: • Has a value of 1 if compiling using the **VERSION1** on page 322 directive (using D40 features). • Has a value of 2 if compiling using the **VERSION2** on page 324 directive (using D45 features). • Has a value of 3 if compiling using the **VERSION3** on page 326 directive (using G06.20 features).
`_GUARDIAN_HOST`	Identifiers used by the C compiler running in the Guardian environment
`_GUARDIAN_TARGET`	Identifiers required for executing in the Guardian environment
`__G_SERIES_RVU`	Declarations that depend on a specific G-series RVU and have the value G06.nn. Defined only for the TNS/R native C/C++ compilers.
`__H_SERIES_RVU`	Declarations that depend on a specific H-series RVU and have the value H06.nn. Defined only for the TNS/E native compilers. For J-series RVUs, use the equivalent H-series RVU.
`__L_SERIES_RVU`	Declarations that depend on a specific L-series RVU and have the value Lyy.mm, where yy is the year and mm is the month that identify the RVU. Defined only for the TNS/X native compilers.
`_IEEE_FLOAT`	Identifiers that support IEEE standard-conforming binary floating-point arithmetic
`_IGNORE_LOCALE`	Identifiers that do not support internationalization and support the C/POSIX locale only
`_OSS_HOST`	Identifiers used by the C compiler running in the OSS environment
`_OSS_TARGET`	Identifiers required for executing in the OSS environment
`__PGO__`	Defined when options `PROFGEN` or `PROFUSE` are specified.

Table Continued

Macro	What It Defines
`_POSIX_C_SOURCE=1`	Identifiers required or permitted by the POSIX.1 standard
`_POSIX_C_SOURCE=2`	Identifiers required or permitted by the POSIX.1 and POSIX.2 standards
`_POSIX_SOURCE`	Identifiers required or permitted by the POSIX.1 standard
`_SQL`	Identifiers required for processing embedded SQL commands by the C compilers; defined only when pragma sql or either flag -Wsql or flag -`Wsqlcomp` is specified on the command line
`_TANDEM_ARCH_`	Identifiers required for use of the TNS, TNS/R, TNS/E or TNS/X architectures. This macro can have these values: **0** indicates the TNS architecture **1** indicates the TNS/R architecture **2** indicates the TNS/E architecture **3** indicates the TNS/X architecture This macro is not defined for the TNS compilers.
`_TANDEM_EXTENSIONS`	Identifiers required for compiling with extensions by the native compilers; defined when pragma `EXTENSIONS` or `-Wextensions` is specified on the command line
`_TANDEM_SOURCE`	Identifiers required or permitted by extensions made by HPE for NonStop systems
`_TNS_X_TARGET`	Identifiers used by the TNS/X native C and C++ compilers; defined only for the TNS/X native C/C++ compilers.
`_TNS_E_TARGET`	Identifiers used by the TNS/E native C and C++ compilers; defined only for the TNS/E native C/C++ compilers.
`_TNS_R_TARGET`	Identifiers used by the TNS/R native C and C++ compilers; defined only for the TNS/R native C/C++ compilers.
`_WIN32_HOST`	Identifiers required for compiling on a PC running the Windows operating system.
`_XOPEN_SOURCE`	Identifiers required or permitted by the XPG4 specification
`_XOPEN_SOURCE_EXTENDED`	Identifiers specified in the XPG4.2 specification as extensions to the XPG4 specification

The _TANDEM_SOURCE macro makes supplementary functions defined by HPE for NonStop systems. If a module compiled for the OSS environment uses functions defined by HPE, you must specify the _TANDEM_SOURCE macro.

The _XOPEN_SOURCE macro makes visible functions defined by either the XPG4 or XPG4 Version 2 specifications. The _XOPEN_SOURCE_EXTENDED macro makes visible XPG4 Version 2 extensions to functions that have base definitions defined in the XPG4 specification.

The _GUARDIAN_TARGET and _OSS_TARGET macros identify a module's execution environment. The compiler uses this information to resolve references to external functions in the C run-time library.

For the TNS compilers, the pragma SYSTYPE GUARDIAN defines the _GUARDIAN_TARGET and _TANDEM_SOURCE macros. For the native compilers, the pragma SYSTYPE GUARDIAN defines only the _GUARDIAN_TARGET macro, while the EXTENSIONS pragma defines the _TANDEM_SOURCE macro. Pragma SYSTYPE OSS defines the _OSS_TARGET and _XOPEN_SOURCE macros.

If the compiler is running in the Guardian environment and SYSTYPE OSS is specified, the compiler undefines _GUARDIAN_TARGET before it defines _OSS_TARGET. If the compiler is running in the OSS environment and SYSTYPE GUARDIAN is specified, the compiler undefines _OSS_TARGET before it defines _GUARDIAN_TARGET.

Do not define or undefine the _GUARDIAN_HOST, _OSS_HOST, _TNS_E_TARGET, and _TNS_R_TARGET macros. These macros are for use by the compiler only.

The _TANDEM_ARCH_ macro can be used to compile code conditionally, depending upon the system architecture used. For example:

```
#if _TANDEM_ARCH_ == 2
printf ("TNS/E\n");
#endif

#if _TANDEM_ARCH_ != 0
printf ("Native system, not TNS\n");
#endif
```

The _IGNORE_LOCALE macro selects macros that support only the C/POSIX locale instead of internationalized functions that support multiple locales. The _IGNORE_LOCALE macro affects functions in the ctypeh or ctype.h header file.

The native compilers provide four additional feature-test macros. Unlike the previously described feature-test macros, these four macros do not determine whether a particular set of features will be included from header files. Instead, these feature-test macros are intended to isolate code by the environment in which it can be run.

The _IEEE_FLOAT macro is automatically defined by the native C and C++ compilers if IEEE floating-point format has been specified. For more details, see the pragma **IEEE_FLOAT** on page 248.

The ENV pragma setting determines which of these native mode feature-test macros is defined:

_COMMON	Identifies code that runs with pragma ENV COMMON set. Pragma ENV COMMON is the default setting. Such code runs in the user code space and requires the Common Run-Time Environment (CRE).
_EMBEDDED	Identifies code that runs with pragma ENV EMBEDDED set. Such code runs in the system code space and cannot use the Common Run-Time Environment (CRE).

Table Continued

_LIBRARY	Identifies code that runs with pragma ENV LIBRARY set. Such code runs in the user library space and requires the Common Run-Time Environment (CRE).
_LIBSPACE	Identifies code that runs with pragma ENV LIBSPACE set. Such code runs in the user library or system library space and cannot use the Common Run-Time Environment (CRE).

These feature-test macros cannot be used to ensure that a C run-time library call can be made in a particular environment. For more details, see the description of pragma **ENV** on page 221.

Preprocessor Operators

There are two preprocessor operators, # and ##. The # operator allows you to create a character string literal when the operator precedes a formal parameter in a macro definition. The ## operator allows you to concatenate two tokens in a macro definition. Both of these operators are used in the context of the #define directive.

Operator

The unary operator # is used only with function-like macros, which take arguments. If the # operator precedes a formal parameter in the macro definition, the actual argument passed by the macro invocation is enclosed in quotation marks and treated as a string literal. The string literal then replaces each occurrence of a combination of the # operator and the formal parameter within the macro definition.

White space preceding the first token of the actual argument and following the last token of the actual argument is deleted. If a character contained in the argument normally requires an escape sequence when used in a string literal, for example the quotation mark (") or backslash (\) characters, the necessary escape backslash is automatically inserted before the character.

Example

```
#define str(x) # x
```

causes the invocation:

```
str(testing)
```

to be expanded into:

```
"testing"
```

Operator

The binary operator ## is used in both object-like and function-like macros. It allows you to concatenate two tokens, and therefore it cannot occur at the beginning or at the end of the macro definition.

If a formal parameter in a macro definition is preceded or followed by the ## operator, the formal parameter is immediately replaced by the unexpanded actual argument. Macro expansion is not performed on the argument prior to replacement. Then, each occurrence of the ## operator in the replacement-list is removed, and the tokens preceding it and following it are concatenated. The resulting token must be a valid token. If it is, the resulting token is available for further macro replacement.

Example

```
#define debug(s, t) printf("x" # s "= %d, x" # t "= %s",\
                              x ## s, x ## t )
```

```
main ()
{
   debug(1, 2);
}
```

After the preprocessor pass, you have:

```
int main(void)
{
...printf("x1= %d, x2= %s", x1, x2);
```

Compiler Pragmas

Compiler pragmas enable you to control various elements of compiler listings, code generation, and building of the object file.

Specify pragmas in these ways, using the method that is appropriate to the compiler and platform you are using:

- In the TACL RUN command that you enter to run the native C and C++ compilers, the TNS C compiler, and Cfront.
- In the G-series TNS c89 utility accompanied by a -Wccom flag.
- In the native c89 or c99 utility accompanied by a flag that corresponds to the pragma (For descriptions of the individual flags, see the c89(1) or c99(1) reference page either online or in the *Open System Services Shell and Utilities Reference Manual.*
- For ETK, the pragmas are set at the project level "Configuration Dependent Properties". It can be accessed within the project properties dialog at the location "Configuration Properties->C/++".

In addition, some pragmas can be specified in the source file itself. Descriptions of the pragmas in this section describe the methods for specifying each pragma. In most cases, pragmas in the source file take precedence over the pragmas in the RUN command or specified to the c89 or c99 utility. For more details, see the individual pragma descriptions.

The compiler version and SYSTYPE setting determine the default values of pragmas. The compilers in the Guardian environment default to SYSTYPE GUARDIAN. The compilers in the OSS environment, which are invoked through the c89 utility or the c99 utility, default to SYSTYPE OSS. For more details, see **SYSTYPE**.

The following table summarizes the compiler pragmas and their descriptions.

Table 27: Compiler Pragma Descriptions

Pragma	Purpose
ALLOW_CPLUSPLUS_COMMENTS	Directs the native mode C compiler to allow the use of comments entered in the style of the C++ language.
ALLOW_EXTERN_EXPLICIT_INSTANTIATION	Specifies an extern to be applied to an explicit template instantiation.
ANSICOMPLY	Directs the TNS C compiler to perform strict syntax checking for compliance to the ISO/ANSI C standard.
ANSISTREAMS	Specifies that Guardian files created by the program, including the standard files (standard input, standard output, and standard error), are odd-unstructured disk files with a file code of 180.

Table Continued

Pragma	Purpose
BASENAME	Specifies that when the executable object program is run, only the base part of the source file name is included in the raw data file. This option is intended for use only with the PROFGEN option and is supported by the TNS/E and TNS/X native compilers.
BITFIELD_CONTAINER	Directs the compiler to accept bitfields larger than 32-bits or whether `types` other than `ints` can be used as `BITFIELD CONTAINER`.
BUILD_NEUTRAL_LIBRARY	Directs the TNS/E native compilers to create a C++ library using only those definitions common to the `VERSION2` and `VERSION3` libraries.
C99	Enables the ISO/IEC 9899:1999 features listed in **C99 Full Support**
C99LITE	Enables the ISO/IEC 9899:1999 features listed in **Appendix G: c99Selected Features (C99LITE)**.
C11	Enables the 2011 ANSI C Standard from L16.05 RVU.
CALL_SHARED	Directs the native compilers to generate PIC (Position-Independent Code) (shared code). This is the default for the TNS/E and TNS/E native compilers.
CHECK	Controls the inclusion of run-time error-checking code in the object file.
CODECOV	Directs the TNS/E and TNS/X native compilers to generate instrumented object code for use by the Code Coverage Utility. If profiling pragma is used in the source code, the instrumentation on individual function are controlled by the profiling pragma. For more information, see the profiling pragma in this manual and *Code Coverage Utilities Manual*.
COLUMNS	Specifies the maximum logical line length of the source file.
CPATHEQ	Specifies a file to be included before the compiler begins compiling the source file.
CPPONLY	Causes only the C macro preprocessor to be run.
CSADDR	Directs the TNS C compiler to copy data objects from the current code space into the stack space.

Table Continued

Pragma	Purpose
DEBUG_USES_LINE_DIRECTIVES	Directs the TNS/E native C/C++ compiler to generate adequate debug information for the eInspect native debugger to honor any `#line` directives appearing in the source file.
ENV	Specifies the intended run-time environment of an object file.
ELD(arg)	Specifies arguments to be passed to the `eld` utility.
ERRORFILE	Directs errors and warnings to a specified file.
ERRORS	Directs the compiler to terminate compilation if it detects more than a specified number of errors.
EXCEPTION_SAFE_SETJMP	Enables special processing of Standard C Library API setjmp, which makes it safe to use setjmp/longjmp in a TNS/R C++ exception-handling environment.
EXTENSIONS	Controls source code use of syntax extensions and standard library extensions defined by HPE.
EXTERN_DATA	Specifies whether external data references (object declared `extern`) can use GP-relative addressing.
FIELDALIGN	Controls the component layout of structures for compatibility between TNS and native or native mixed-language structure layout.
FORCE_VTBL	Forces definition of virtual function table in cases where the heuristic used by the native C++ compiler to decide on definition of virtual function tables provides no guidance.
FORCE_STATIC_TYPEINFO	Forces the typeinfo variables to be static to the file.
FORCE_STATIC_VTBL	Forces the virtual function tables that are created by the compiler to be static to the file and not exported.
FUNCTION	Declares attributes of external routines.
GLOBALIZED	Directs the native C and C++ compilers to generate preemptable object code.
HEADERS	Directs the native C and C++ compilers to print a list of included header files.
HEAP	The `HEAP` pragma specifies the maximum heap size of a program compiled with the `RUNNABLE` pragma.

Table Continued

Pragma	Purpose
HIGHPIN	Specifies that the object file should be run at an operating system high PIN (256 or greater) or at a low PIN (0 through 254).
HIGHREQUESTERS	Specifies that the object file supports high PIN requesters if the object file includes the main function.
ICODE	Controls whether the compiler listing includes the instruction-code mnemonics generated for each function immediately following the source text of the function.
IEEE_FLOAT	New at the G06.06 RVU. Specifies that the native C++ compiler is to use the IEEE floating-point format for performing floating-point computations. (`TANDEM_FLOAT` is the TNS/R default mode for floating-point computations, `IEEE_FLOAT` is the TNS/E and TNS/X default.)
INLINE	For TNS C programs, controls whether the compiler generates inline code for certain standard library functions instead of generating a function call. For native C++ programs, controls whether functions declared inline are actually generated inline.
INLINE_COMPILER_GENERATED_FUNCTIONS	Forces all compiler-generated functions to be inlined.
INLINE_LIMIT	Specifies the maximum number of lines that the compiler can inline, where *n* denotes the number of lines as an integer.
INLINE_STRING_LITERALS	Allows the compiler to inline functions that take the address of a string literal.
INNERLIST	Controls whether the compiler listing includes the instruction-code mnemonics generated for each statement immediately following the source text of the statement.
INSPECT	Controls whether the symbolic debugger or the default system debugger is used as the default debugger for the object file.
KR	Specifies to the native C compiler that Kernighan & Ritchie (common-usage) C is being compiled instead of ISO/ANSI Standard C.
LARGESYM	Directs the TNS C compiler and Cfront to generate all the symbols for a given compilation to a single symbols data block containing information used by the Inspect debugger to display information about a variable or to display its contents.

Table Continued

Pragma	Purpose
LD(arg)	Specifies arguments to be passed to the `ld` utility.
LINES	Specifies the maximum number of output lines per page for the compiler listing file.
LINKFILE	Invokes the appropriate linker and specifies a command file to be passed.
LIST	Controls the generation of compiler-listing text.
LMAP	Controls the generation and presentation of load-map information in the compiler listing.
MAP	Controls the generation of identifier maps in the compiler listing.
MAPINCLUDE	Specifies how to transform file names within `#include` directives.
MAXALIGN	Specifies that objects of a composite type are given the maximum alignment supported by the compiler.
MIGRATION_CHECK	Directs the compiler to perform a migration check, to aid in migrating from `VERSION2` to `VERSION3` of the Standard C ++ Library.
MIGRATION_CHECK 32TO64	Enables additional compiler diagnostics. These warnings detect valid C/C++ code that potentially works in an unexpected fashion when code designed for ILP32 is compiled using the LP64 data model.
NEST	Controls whether the TNS compiler accepts nested comments.
NEUTRAL	Specifies a `struct`, `class`, or `union` definition and marks it as being available in the neutral C++ standard library on TNS/E and TNS/X systems.
NLD(arg)	Specifies arguments to be passed to the `nld` utility.
NOEXCEPTIONS	Disables support for exception handling by the native C++ compiler when you are also using the VERSION2 or VERSION3 directive.
NON_SHARED	Directs the compiler to generate code that is not PIC (Position-Independent Code), that cannot be shared and that cannot access PIC files. `NON_SHARED` is the default for the TNS/R native compilers but is not allowed for the TNS/E and TNS/X native compilers.

Table Continued

Pragma	Purpose
OLDCALLS	Controls how the TNS C compiler generates code for function calls.
OLIMIT	Specifies the maximum decimal number of basic blocks of a routine to be optimized by the global optimizer.
ONCE	Specifies that the file containing this pragma will be compiled only once during the compilation.
OPTFILE	Specifies an optimizer file, which contains a list of functions that are to be optimized at the level specified in the file.
OPTIMIZE	Controls the level to which the compiler optimizes the object code.
OVERFLOW_TRAPS	Determines whether the native C and C++ compilers generate code with arithmetic overflow traps.
PAGE	Causes a page eject in the compiler listing and prints a page heading.
POOL_STRING_LITERALS	Specifies that within a compilation unit multiple occurrences of the same string literal are to occupy the same storage space.
POP	Directs the native compilers to restore the value of certain pragmas that were stored earlier by a `PUSH` pragma.
p32_shared	Supports ILP32 data in the LP64 data model compiler. This pragma is available in the native C and C++ compilers. It can be specified only in the source file using the `#pragma p32_shared` syntax. For more information, see **LP64 Data Model”**.
p64_shared	It indicates within the definition, that the unqualified pointer declarations are 64-bit pointers – regardless of the data model. For more information, see **LP64 Data Model” (page 415).**.
PROFDIR	Specifies where an instrumented object file is to create the raw data file. This option is intended for use only with the `PROFGEN` or `CODECOV` option, and is supported only by the TNS/E native compilers.
PROFILING/NOPROFILING	Directs the TNS/X native compiler to perform profiling related code instrumentation or optimization on individual functions.

Table Continued

Pragma	Purpose
PROFGEN	Directs the TNS/E and TNS/X native compilers to generate an instrumented object file for use in profile-guided optimization. If profiling pragma is used in the source code, the instrumentation on individual function are controlled by the profiling pragma. For more information, see the profiling pragma in this manual and *Code Coverage Utilities Manual*.
PROFUSE	Directs the TNS/E and TNS/X native compilers to generate optimized object code based on information in a dynamic profiling information (DPI) file. If profiling pragma is used in the source code, the optimization on individual function are controlled by the profiling pragma. For more information, see the profiling pragma in this manual and *Code Coverage Utilities Manual*.
PUSH	Directs the compiler to save the value of certain pragmas.
REFALIGNED	Specifies the default reference alignment for pointers in native C and C++ programs.
REMARKS	Directs the native C and C++ compilers to issue remarks.
RUNNABLE	Directs the Guardian compiler to generate an executable object file instead of a linkable object file for a single-module program.
RUNNAMED	Specifies that the object file runs as a named process.
RVU	Sets the value of the `_H_SERIES_RVU` or `_G_SERIES_RVU` feature-test macro.
SAVEABEND	Controls whether the system creates a save file if the program terminates abnormally during execution.
SEARCH	Directs the Binder or linker to search a given object file when attempting to resolve external references in a program compiled with the `RUNNABLE` pragma.
SECTION	Gives a name to a section of a source file for use in an `#include` directive.
SHARED	Directs the compiler to generate PIC (Position-Independent Code) that can be shared, and to invoke the `ld`, `eld`, or `xld` linker to create a PIC library file or a dynamic-link library (DLL).

Table Continued

Pragma	Purpose
SQL	Enables the compiler to process subsequent SQL statements.
SQLMEM	Provides the ability to alter the placement of SQL data structures from extended memory to the user data segment.
SRL	Specifies that a module is being compiled to link into a native user library. This pragma is valid only for TNS/R compilers.
SRLExportClassMembers	Controls which members of a class are exported from an SRL or user library (a private shared run-time library). This pragma is valid only for TNS/R compilers.
SRLName	Is used to "name" an SRL or user library (a private shared run-time library). This pragma is valid only for TNS/R compilers.
SSV	Specifies a list of search subvolumes (SSVs) to be searched for files specified in `#include` directives.
STDFILES	Controls the automatic opening of the three standard files; `stdin`, `stdout`, and `stderr`.
STRICT	Directs the TNS C compiler or Cfront to generate a warning if it encounters one of a number of valid, but questionable, syntactic or semantic constructs.
SUPPRESS	Controls the generation of compiler-listing text, regardless of the status of the `LIST` pragma.
SUPPRESS_VTBL	Suppresses the definition of virtual function tables in cases where the heuristic used by the native C++ compiler to decide on definition of virtual function tables provides no guidance.
SYMBOLS	Controls the inclusion of symbol information in the object file for use by a symbolic debugger.
SYNTAX	Directs the compiler to not generate an object file but merely to check the source text for syntactic and semantic errors.
SYSTYPE	Controls whether the generated code’s target execution environment is the NonStop environment.

Table Continued

Pragma	Purpose
TANDEM_FLOAT	Specifies that the native C++ compiler is to use Tandem floating-point format for performing floating-point computations. TANDEM_FLOAT is the default mode for G-series floating-point computations.
TARGET	Directs the native compilers to create an output file for the specified target hardware platform.
TEMPEXTENT	Controls the size of the temporary files created by the compiler.
TRIGRAPH	Controls whether the TNS C compiler or Cfront translate trigraphs for the current compilation.
VERSION1	Directs the TNS/R native C++ compiler to compile according to the D40 version or dialect of C++. Disables all new features added from the D45 RVU onward. VERSION1 is the default compilation mode for D45 and all RVUs until the G06.20 RVU.
VERSION2	Directs the native C++ compiler to compile using the dialect or features available beginning with the D45 version of the HPE C++ language.
VERSION3	Directs the native C++ compiler to compile according to the G06.20 version or dialect of C++. Enables all new features added at the G06.20 RVU and enforces the ISO/IEC IS 14882-1998 standard. VERSION3is the default compilation mode for G06.20 and all subsequent RVUs. Compare with VERSION2 and VERSION1.
VERSION4	Directs the native C++ compiler to compile according to ISO/IEC IS 14882-2011 standard. Programs must be executed on L17.02 RVU or later. Supported only by TNS/X. Compare with VERSION3, VERSION2, and VERSION1.
WARN	Controls the generation of all or selected warning messages.
WIDE	Specifies the data model, which defines the size of the data type int.
XLD(arg)	Specifies arguments to be passed to the xld utility.
XMEM	Controls which TNS memory model, large or small, the object file uses.
XVAR	Controls whether the TNS C compiler places subsequent global or static aggregates in extended memory or in the user data segment.

ALLOW_CPLUSPLUS_COMMENTS

The ALLOW_CPLUSPLUS_COMMENTS pragma directs the native mode C compiler to allow the use of comments entered in the style of the C++ language.

```
ALLOW_CPLUSPLUS_COMMENTS
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS c89 utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set	Not set
Native c89 and c99 utilities	Not set	Not set
TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

- ALLOW_CPLUSPLUS_COMMENTS is a command-line directive that must be entered on the compiler RUN command line, not in the source text.
- The ALLOW_CPLUSPLUS_COMMENTS directive can also be specified with the -Wallow_cplusplus_comments flag of the c89 or the c99 utility.
- When you use the ALLOW_CPLUSPLUS_COMMENTS directive, your native mode C program can include the comment delimiter that is standard in C++, namely, the double slash (//).

Examples

```
//  C++ comment (to end of line)
/*  C comment (between delimiters;
    can span several lines)  */
```

ALLOW_EXTERN_EXPLICIT_INSTANTIATION

The `ALLOW_EXTERN_EXPLICIT_INSTANTIATION` command-line option specifies an `extern` to be applied to an explicit template instantiation. This option will suppress the instantiation of the template. The default behavior is to instantiate the template.

```
ALLOW_EXTERN_EXPLICIT_INSTANTIATION
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set	Not set

Native
`c89`
and
`c99`
utilities
	Not set	Not set

TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	N.A.	N.A.

Usage Guidelines

`ALLOW_EXTERN_EXPLICIT_INSTANTIATION` can be entered on the compiler RUN command line (NMCPLUS) or be specified with the `-Wallow_extern_explicit_instantiation` flag of the `c89` or the `c99` utility.

ANSICOMPLY

The `ANSICOMPLY` pragma directs the TNS C compiler to perform strict syntax checking for compliance to the ISO/ANSI C standard.

```
ANSICOMPLY
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	Not set	Not set
G-series TNS c89 utility	Not set	Not set
TNS/R native C and C++ compilers	N.A.	N.A.
Native c89 and c99 utilities	N.A.	N.A.
TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	N.A.	N.A.

Usage Guidelines

- In the Guardian TNS C environment, the `ANSICOMPLY` pragma can appear either on the compiler RUN command line or in the source text. However, in the G-series OSS TNS C environment, the ANSICOMPLY pragma must be specified in the source file. In either environment, if you use the pragma within a source file, the pragma must appear before any C language source statements.
- The `ANSICOMPLY` pragma also sets the `ANSISTREAMS` pragma,
- For ISO/ANSI C standard compliance checking, the compiler:
  - Disables arithmetic overflow traps.
  - Flags HPE C language extensions for NonStop systems.
  - Checks for conformance to the `#define` preprocessor directive. For a description of the difference between HPE C and the ISO/ANSI C standard, see **Converting C-Series TNS Programs to Use the Current TNS Compiler** on page 172.
- By default, the native C and C++ compilers conform to the ISO/ANSI C standard.

ANSISTREAMS

The `ANSISTREAMS` pragma specifies that Guardian files created by the program, including the standard files (standard input, standard output, and standard error), are odd-unstructured disk files with a file code of 180. The maximum length of a line for type 180 files is the maximum size of a file. If this pragma is not asserted, files are EDIT files with a file code of 101.

This pragma enables the compiler to conform to the ISO/ANSI C standard that requires that the compiler support text files with lines containing at least 254 characters including the terminating new-line character.

```
ANSISTREAMS
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	Not set	`ANSISTREAMS`
G-series TNS `c89` utility	`ANSISTREAMS`	`ANSISTREAMS`
TNS/R native C and C++ compilers	`Not set`	`ANSISTREAMS`

Native
`c89`
and
`c99`
utilities
	`ANSISTREAMS`	`ANSISTREAMS`

TNS/E native C and C++ compilers	`Not set`	`ANSISTREAMS`
TNS/X native C and C++ compilers	`Not set`	`ANSISTREAMS`

Usage Guidelines

- `ANSISTREAMS` is a command-line directive that must be entered on the compiler RUN command line, not in the source text.
- The `ANSISTREAMS` directive can also be specified with the -`Wallow_cplusplus_comments` flag of the `c89` or the `c99` utility.
- A text file opened with the `ANSISTREAMS` pragma in effect cannot be manipulated with the `edfseek()`, `edftell()`, or `edlseek()` functions.
- Without the `ANSISTREAMS` pragma, a text line can contain as many as 239 characters.
- The `ANSISTREAMS` pragma does not affect I/O performed by Guardian procedures.

BASENAME

Information saved in raw data files used for code profiling includes the names of the instrumented source files used in the creation of those raw data files. By default, these are fully qualified source file names, following the rules for fully qualified names for the respective host platforms. The BASENAME option specifies that when the executable object program is run, only the base part of the source file name will be included in the raw data file. The BASENAME option makes it possible for source files compiled with the PROFGEN option and the same source files compiled with the PROFUSE option to be in different locations.

The BASENAME option is intended for use only with the PROFGEN option and is supported only by the TNS/E native compilers.

```
BASENAME
```

For more information about the code profiling and the use of the BASENAME and PROFUSE options, see the *Code Profiling Utilities Manual*.

The BASENAME default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	N.A.	N.A.

Native
`c89`
and
`c99`
utilities
	Not set	Not set

TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

- The `BASENAME` option can be entered only on the compiler RUN command. It can also be specified with the `-Wbasename` flag in the `c89` or `c99` command in the OSS and Windows environments.
- In the OSS and Windows environments, the `–Wbasename` flag has an effect only for `–Wtarget=ipf` or `–Wtarget=tns/e` . It is ignored (and no diagnostic is issued) for `–Wtarget=mips` or `-Wtarget=tns/r`.
- If an application is compiled with the BASENAME option, then when the application is recompiled with the PROFUSE option, the BASENAME option should also be specified; otherwise, no code profiling is done and the compiler issues a warning message.
- The BASENAME option is not allowed with the CODECOV option. If BASENAME and CODECOV are specified on the same command line, a warning message is issued.

BITFIELD_CONTAINER

The `BITFIELD_CONTAINER` pragma provides the user a way to specify whether non-standard types are allowed as bitfield containers.

`-Wbitfield_container =`**value** (OSS and Windows platform) and `bitfield_container =`**value** (Guardian platform)

value is defined as one of the following modes:

`int`

In this mode, bitfields are packed into 32-bit `ints`. The compiler does not accept bitfields larger than 32-bits. The compiler gives an error for any bitfield declared to be of `long long` type. (Unless – Wextensions are also specified.) The compiler returns a warning for other non-standard integer types.

`long`

In this mode, bitfields whose base type is larger than 32-bit, are packed into 64-bit `ints`. All other bitfields are packed into 32-bit `ints`. The compiler accepts `long long` and `long` bitfield types. It also accepts up to 64-bits in length. The compiler continues to return a warning for other non-standard integer types.

`all`

In this mode, all bitfields are packed into `ints` defined by their base type. The compiler accepts any integer type for a bitfield. This provides compatibility with how some other compilers pack bitfields. To see how the use of `all` can change the packing of a structure, consider this struct:

```
struct {
    short int a : 10;
    short int b : 10;
    short int c : 10;
} s;
```

With `int` or `long` containers, the compiler tries to pack the bitfields into 32-bit `ints.` Hence, all three bitfields fit into one 32-bit `int`, the size of the `struct` is 4 bytes.

With `all` containers, the bitfields are packed into 16-bit (the size of a short `int`) containers. The first bitfield fits into the first 16-bit `int`, but the second one does not, and requires a second 16-bit `int` to hold it. Likewise, the third bitfield requires a third 16-bit `int` to hold it. So, the size of this `struct` is 6 bytes.

Usage Guidelines

- The default value of the `BITFIELD_CONTAINER` option is `int` except when `–Wlp64` option is specified in which case, the default is `long`.
- This command-line directive is only supported in TNS/E compilers running on H06.24/J06.13 RVU and later.
- `BITFIELD_CONTAINER` is a command-line directive that must be entered on the compiler RUN command-line, not in the source text.

NOTE: The specified rules, apply to bitfields declared in `AUTO` or `PLATFORM` (default) `structs`. Any bitfield declared in a `SHARED2 struct` continues to follow the `SHARED2` rules. Also, bitfields in `SHARED2 structs` cannot be larger than 32-bit. Bitfields declared in `SHARED8 structs` can be larger than 32-bits (if the `BITFIELD_CONTAINER` is `long` or `all`). The compiler returns a warning in this case. Bitfields in `SHARED8 structs` are packed, but the bitfields larger than 32-bits are packed into 64-bit containers.

BOOST

The BOOST command line option indicates that NonStop supported Boost libraries are used. The BOOST command line option performs the following:

- Defines the macro __NSK_BOOST_VERSION__ and assigns a value of 156. This macro cannot be explicitly defined by the user using either the -D command line option or the #define preprocessor directive. The main Boost config header returns an error if the value of __NSK_BOOST_VERSION__ does not match its version. The value 156 corresponds to Boost version 1.56.

  NOTE: You cannot undefine the __NSK_BOOST_VERSION__ macro by using –U command line option or the #undef preprocessor directive.

- Adds -I $COMP_ROOT/usr/boost_1_56 to the list of directories to search for headers.
- Adds the appropriate libboost archive to the set of objects supplied to the linker if c89 or c99 performs a link step.

  NOTE: Programs that use the test library and require a test library archive must explicitly name the archive in link step. For example, ${COMP_ROOT}/usr/boost_1_56/libv4/libboostw_test_exec_monitor.a.

- Shortens mangled names to 2048 characters, but only for names that depend upon entities that are defined in boost namespace. CRC encoding is used to insure that the shortened names are unique. However, names that are shortened cannot be de-mangled.

  Boost mangled names tend to be excessively large, many times exceeding 10000 characters. Shortening the names uses fewer resources during compilation and uses less space in the resulting object file. For example, in the proc_info section.

Usage Guidelines

The BOOST pragma is not available on Guardian and OSS/Windows, the form is -Wboost.

BUILD_NEUTRAL_LIBRARY

The BUILD_NEUTRAL_LIBRARY pragma is a command-line directive and directs the native compilers to create a C++ library using only those components common to the VERSION2 and VERSION3 standard libraries.

```
BUILD_NEUTRAL_LIBRARY
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS c89 utility	N.A.	N.A.

Table Continued

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS/R native C and C++ compilers	N.A.	N.A.

Native
`c89`
and
`c99`
utilities
	G-series: N.A. H-series and J-series: not set	G-series: N.A. H-series and J-series: not set

TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

- `BUILD_NEUTRAL_LIBRARY` is a command-line directive that must be entered on the compiler RUN command line, not in the source text.
- The `BUILD_NEUTRAL_LIBRARY` directive can also be specified with the `-Wbuild_neutral_library` flag of the `c89` or the `c99` utility.

  This flag is valid only for TNS/E-targeted C++ compilations and only when the -Wversion2 or -Wversion3 flag is also used.

- For other guidelines, see **Using the Neutral C++ Dialect** on page 102.

Example

See **Using the Neutral C++ Dialect** on page 102.

C99LITE

The C99LITE pragma, supported for H06.08 and later H-series RVUs and J06.03 and later J-series RVUs only, enables the subset of features from the 1999 update to the ISO/IEC C standard (ISO/IEC 9899:1999) supported by HPE C. These features are summarized in **c99 Selected Features (C99LITE)** on page 598.

```
C99LITE
```

NOTE: The complete set of features from the 1999 update to the ISO/IEC C standard (ISO/IEC 9899:1999) are supported for TNS/E-targeted programs compiled on systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs. For more information, see **c99 Full Support** on page 602.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS c89 utility	N.A.	N.A.
TNS/R native C and C++ compilers	N.A.	N.A.
Native c89 utility	Not enabled	Not enabled
c99 utility	N.A.	N.A.
TNS/E native C and C++ compilers	Not enabled	Not enabled
TNS/X native C and C++ compilers	Not enabled	Not enabled

Usage Guidelines

- In the Guardian environment, the `C99LITE` pragma is recognized only on the CCOMP command; it is ignored (and no diagnostic is issued) on the CPPCOMP command. The OSS and Windows equivalent, `–Wc99lite`, can appear only on the command line for the `c89` utility; it is recognized only when compiling a C program; it is ignored (and no diagnostic is issued) when compiling a C++ program.
- In the Guardian environment, neither the SQL nor the KR pragmas can be specified along with the `C99LITE` pragma. In the OSS and Windows environments, neither the `–Wsql` nor the `–Wkr` options can be specified along with the `-Wc99lite` option.
- The `–Wc99lite` option is effective only when `–Wtarget=ipf` or `-Wtarget=tns/e` is specified (or implied). If `–Wtarget=mips` or `-Wtarget=tns/r` is specified, a warning message is issued.

CALL_SHARED

The `CALL_SHARED` pragma directs the native C and C++ compilers to generate shared code, which is PIC (Position-Independent Code). The loadfile that results can access PIC library files (DLLs).

Compare the action of `CALL_SHARED` with that of the two related pragmas:

- **NON_SHARED** on page 275 directs the compiler to generate a TNS/R non-PIC loadfile that cannot be shared and cannot access PIC files.
- **SHARED** on page 300 directs the compiler to generate PIC and to invoke the `eld` or `ld` linker to create a PIC library file.

  `CALL_SHARED`

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	`NON_SHARED`	`NON_SHARED`

Native
`c89`
and
`c99`
utilities
	TNS/R code: `NON_SHARED`
	TNS/E code: `CALL_SHARED`
	TNS/X code: `CALL_SHARED`
		TNS/R code: `NON_SHARED`
		TNS/E code: `CALL_SHARED`
		TNS/X code: `CALL_SHARED`

TNS/E native C and C++ compilers	`CALL_SHARED`	`CALL_SHARED`
TNS/X native C and C++ compilers	`CALL_SHARED`	`CALL_SHARED`

Usage Guidelines

- The `CALL_SHARED` pragma can appear only on the RUN command line for NMC or NMCPLUS (G-series Guardian) or for CCOMP or CPPCOMP (L-series or H-series or J-series Guardian). The OSS equivalent (`-Wcall_shared`) can appear only on the command line for the `c89` or the `c99` utility.
- On Guardian environment, the default behavior of `CALL_SHARED` is to not call the linker. The native C or C++ driver only calls the linker if you also specify either **RUNNABLE** on page 292 or **LINKFILE** on page 262, or if you specify **SHARED** on page 300.
  - if you specify `CALL_SHARED`, the compilation results in a PIC linkable object file (a PIC linkfile).
  - If you include both the `CALL_SHARED` and `RUNNABLE` pragmas, the compilation results in a PIC executable object file (a PIC loadfile) from the linker.
- In the OSS file system, the default behavior of the `-Wcall_shared` flag is to automatically call the `eld` or `ld` linker.

    - If you specify -Wcall_shared, the compilation results in a PIC executable object file (a PIC loadfile).
    - If you specify the -c option with -Wcall_shared, the linker is not called, and the result is a PIC linkable object file (a PIC linkfile).

- If you specify the CALL_SHARED pragma, the compiler driver passes these to the linker:
    - For a TNS/R program, CCPPMAIN (Guardian) or ccppmain.o (On OSS environment or PC)
    - For a TNS/E program, CCPLMAIN (Guardian) or ccplmain.o (On OSS environment or PC)
    - For a TNS/X program, CCPMAINX(Guardian) or ccpmainx.o (On OSS environment or PC)
    - For a TNS/R C++ version 2 program, CPPINIT2 (Guardian) or cppinit2.o (On OSS environment or PC)
    - For TNS/R C++ version 3 program, CPPINIT4 (Guardian) or cppinit4.o (On OSS environment or PC)
- The CALL_SHARED and SHARED pragmas cannot be used with the **SRL** on page 305 or **NON_SHARED** on page 275 pragmas. A warning is issued if these pragmas are combined.
- **EXTERN_DATA** on page 230 gp_ok is not compatible with generation of shared code. The compiler issues a warning if this pragma is combined with the **SHARED** on page 300 pragma or **CALL_SHARED** on page 210, and ignores theGP_OK directive.
- For complete information about programming with DLLs and linker options for DLLs, see the:
    - *DLL Programmer’s Guide for TNS/R Systems*
    - *DLL Programmer’s Guide for TNS/E and TNS/X Systems*
    - *eld and xld Manual*
    - *ld Manual*
    - *rld Manual*

CHECK

The CHECK pragma controls the inclusion of run-time error-checking code in the object file. The CHECK pragma directs the TNS C compiler to include these run-time checks, and NOCHECK directs it to omit them.

```
[NO]CHECK
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	CHECK	NOCHECK
G-series TNS c89 utility	CHECK	NOCHECK
TNS/R native C and C++ compilers	N.A.	N.A.
Native c89 and c99 utilities	N.A.	N.A.
TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	N.A.	N.A.

Usage Guidelines

- In the Guardian environment, the CHECK pragma must be entered on the compiler RUN command line. In the OSS environment, the CHECK pragma must be entered in the source file.
- If one of the run-time checks controlled by the CHECK pragma discovers an inconsistency, it prints a stack trace to the standard error file and terminates program execution.
- The run-time checks diagnose several inconsistencies, including:
  - Attempted conversion of a byte pointer to a word pointer when the byte pointer points to an odd byte address
  - Attempted conversion of a 32-bit pointer to a 16-bit pointer when the 32-bit pointer points to an address in extended memory
  - Library calls that do not specify valid values for arguments
- The run-time checks provided by the CHECK pragma can slow the execution of your program. Consequently, you should use CHECK only when developing and debugging your program.
- The native C and C++ compilers do not support these pragmas. The native C run-time library does not provide the additional parameter checking provided by the TNS C run-time library.

CODECOV

The CODECOV command line option directs the TNS/E or TNS/X C and C++ compilers to generate object code containing special instrumentation for use by the Code Coverage Utility.

```
CODECOV
```

The instrumentation produced by the CODECOV option consists of extra code that records which functions and blocks are executed, and how many times each is executed. The Code Coverage Utility uses this information to produce a report indicating what code in a program file or DLL was actually executed during one or more invocations. The code coverage report is a set of HTML files that you can view with any standard HTML browser. For more information about the Code Coverage Utility, see the *Code Profiling Utilities Manual*.

The code coverage on individual functions are also controlled by “#pragma profiling” and “#pragma noprofiling”. For more information, see **PROFILING/NOPROFLING** on page 286.

The CODECOV default settings are:

	SYSTYPE GUARDIAN	**SYSTYPE OSS**
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	N.A.	N.A.

Native
`c89`
and
`c99`
utilities
	Not set	Not set

TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

- The `CODECOV` option can be entered only on the compiler RUN command. It can also be specified with the -`Wcodecov` flag on the `c89` or `c99` command in the OSS and Windows environments.
- In the OSS and Windows environments, the - `Wcodecov` flag has an effect only for `–Wtarget=ipf` or `–Wtarget=tns/e` or `–Wtarget=tns/x`. It is ignored (and no diagnostic is issued) for – `Wtarget=mips` or -`Wtarget=tns/r`.
- Instrumented object code can significantly increase both the compile time and execution time required by a program. Therefore, the `CODECOV` option should be used only in a test environment.

COLUMNS

The `COLUMNS` pragma specifies the maximum logical line length of the source file.

```
COLUMNS last-column
```

last-column

specifies the last column in a source line to process. The compiler ignores any text in the line beyond this column. *last-column* must be in the range 20 through 32767.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	Not set	Not set
G-series TNS `c89` utility	Not set	Not set
TNS/R native C and C++ compilers	Not set	Not set

Native
`c89`
and
`c99`
utilities
	Not set	Not set

TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

- The `COLUMNS` pragma can be entered on the compiler RUN command line or in the source text. If it is included in the source file, the `COLUMNS` pragma must appear before any `SECTION` pragmas.
- The `OLUMNS` pragma can also be specified with the -`Wcolumns=`*`c`* flag of the `c89` or the `c99` utility.
- If you do not use the `COLUMNS` pragma, the compiler processes the full length of each source line.
- The `COLUMNS` pragma can be specified only once in each source file and it must be the only text in the source line.
- The `COLUMNS` pragma causes physical source lines after *last-column* to be truncated before the C preprocessor begins the translation phase.
- The `COLUMNS` pragma currently in effect depends on the context:

- The main input file uses the *last-column* value specified in a `COLUMNS` pragma. If no `COLUMNS` pragma is specified, the compiler processes the full length of each source line.
- At each `#include` directive, each included file initially assumes the *last-column* value in effect when the `#include` directive appeared. If a `COLUMNS` pragma is specified in the included file, the compiler uses the *last-column* value specified.
- After a `#include` directive completes execution (that is, after the end of file is reached), the compiler restores the *last-column* value to what it was when the `#include` directive appeared.
- In all other cases, the *last-column* value is set by the most recently processed `COLUMNS` pragma.

CPATHEQ

The `CPATHEQ` pragma specifies a file to be included before the compiler begins compiling the source file. The `CPATHEQ` file is an EDIT file that typically contains `MAPINCLUDE` pragmas to submit to the C or C++ compiler. The CPATHEQ file can also be used to provide command-line options that exceed a single command line, such as SSV pragmas; a `CPATHEQ` file that contains command-line options is not treated as part of the command line but as a source file.

```
CPATHEQ [ "file-name" ]
```

file-name

identifies a `CPATHEQ` file.

The pragma default settings are:

	SYSTYPE GUARDIAN	**SYSTYPE OSS and PC**
TNS C compiler	Not set	Not set
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set	N.A.

Native
`c89`
and
`c99`
utilities
	N.A.	N.A.

TNS/E native C and C++ compilers	Not set	N.A.
TNS/X native C and C++ compilers	Not set	N.A.

Usage Guidelines

- The CPATHEQ pragma can appear only on the compiler RUN command line for the native C and C++ compilers. For the TNS C compiler and Cfront, the CPATHEQ pragma can appear either on the compiler RUN command line or in the source file.
- If *file-name* is omitted, the native compiler searches for a file named CPATHEQ in the same subvolume as the compiler. Prior to the D45 RVU, the native compilers searched in the default subvolume of the source file. The TNS compilers also search in the default subvolume.
- For additional guidelines and examples of using a CPATHEQ file with class libraries such as Tools.h++, see **Pragmas for Tools.h++** on page 108.

CPPONLY

The CPPONLY pragma causes only the C macro preprocessor to be run.

```
CPPONLY [ (option [, option] ) ]

option:
  { [no]comments | [no]lines | file "file-name" }
```

[no]comments

specifies whether comments are preserved in the preprocessed file.

[no]lines

specifies whether #line directives are generated in the preprocessed file.

file "*file-name*"

specifies a Guardian type-180 file to receive the preprocessed output. Quotation marks are required delimiters around the file name.

The default *option* is comments, lines.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS c89 utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set	Not set

Table Continued

	SYSTYPE GUARDIAN	SYSTYPE OSS
Native c89 and c99 utilities	N.A.	N.A.
TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

- The CPPONLY pragma can appear only on the compiler RUN command line.
- The CPPONLY pragma can be specified only when running the compiler in the Guardian environment.
- If you do not include the file option, the preprocessed output is written to the OUT file that was specified in the compiler RUN command. In this case, if the OUT file is an EDIT file, preprocessed output lines might get truncated. To prevent this problem, you should include the file option and specify a file of type 180 (a C disk file, an odd-unstructured file).

CSADDR

The CSADDR pragma directs the TNS C compiler to copy data objects from the current code space into the stack space. The CSADDR pragma enables you to pass an address pointing at the current code space to a function of a different code space. The CSADDR pragma is intended for functions compiled under the ENV LIBRARY or ENV LIBSPACE pragma.

```
CSADDR
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	Not set	Not set
G-series TNS c89 utility	Not set	Not set
TNS/R native C and C++ compilers	N.A.	N.A.

Table Continued

	SYSTYPE GUARDIAN	SYSTYPE OSS
Native c89 and c99 utilities	N.A.	N.A.
TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	N.A.	N.A.

Usage Guidelines

- The `CSADDR` pragma must be entered in the source file.
- If you specify the `CSADDR` pragma, the compiler allocates sufficient space on the stack for each code space data object to which a function argument points. It generates move instructions to copy each data object to the corresponding implicit stack address, based on information from previous data declarations. The new stack address is then passed as the function argument.
- For examples and additional guidelines on using the `CSADDR` pragma, see **System-Level Programming** on page 166.
- The native C and C++ compilers do not support this pragma. The native process memory architecture does not require its use.

DEBUG_USES_LINE_DIRECTIVES

The `DEBUG_USES_LINE_DIRECTIVES` pragma is a command line directive and directs the TNS/E native C/C++ compiler to generate adequate debug information for the eInspect native debugger to honor any `#line` directives appearing in the source file. This pragma can if you are compiling a preprocessed output file that contains #line directives and you want the debugger to map the source file based on the `#line` directives.

```
DEBUG_USES_LINE_DIRECTIVES
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A	N.A
G-series TNS c89 utility	N.A	N.A
TNS/R native C and C++ compilers	N.A	N.A
Native c89 and c99 utilities	Not set	Not set

Table Continued

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

- The `DEBUG_USES_LINE_DIRECTIVES` is a command line pragma and must not be entered in the source text.
- The `DEBUG_USES_LINE_DIRECTIVES` pragma must be entered on the compiler RUN command line (CCOMP or CPPCOMP) or specified with the `WDEBUG_USES_LINE_DIRECTIVES` option of the c89 or c99 utility.
- By default, the TNS/X native C/C++ compiler generates the required debug information to honor `#line` directives. Therefore, this pragma is not required to be specified for TNS/X C/C++.

ELD(arg)

The `ELD` pragma specifies arguments to be passed to the `eld` utility, the TNS/E linker for PIC (Position-Independent Code).

```
ELD(arg)
```

arg

is any argument accepted by the `eld` utility.

For more details on valid syntax and semantics, see the *eld and xld Manual*.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	N.A.	N.A.

Table Continued

	SYSTYPE GUARDIAN	SYSTYPE OSS

Native
`c89`
and
`c99`
utilities
	Not set	Not set

TNS/E native C and C++ compilers	Not set	Not set

Usage Guidelines

- The `ELD` pragma is a command-line directive and must not be entered in the source text.
- On Guardian environment, the `ELD` pragma must be entered on the compiler RUN command line for TNS/E native C/C++. On OSS environment, specify the `ELD` pragma by using the `-Weld=`*`arg`* flag for the `c89` or the `c99` utility.
- The `ELD` pragma does not actually invoke the `eld` linker. To invoke `eld`, you must include other pragmas such as **RUNNABLE** on page 292, **SHARED** on page 300 or **LINKFILE** on page 262. If `eld` is not invoked, this pragma is ignored.
- This TNS/E native C command example shows the `ELD` pragma. Note that if you specify `SHARED`, no need to include `RUNNABLE`:

  ```
  ccomp /in prog1/ prog1o; runnable, eld(-set floattype neutral)
  ```

ENV

The `ENV` pragma specifies the intended run-time environment of an object file.

```
ENV env-option

env-option:
   { COMMON    }
   { EMBEDDED  }
   { LIBRARY   }
   { LIBSPACE  }
```

`COMMON`

specifies that the module requires the Common Run-Time Environment (CRE).

The module can be called by routines written in any language that runs in the CRE. Use this option for user code functions that run in the CRE.

For the native C and C++ compilers, this option sets the `_COMMON` feature-test macro.

`EMBEDDED`

specifies that the module does not require resources provided by the CRE and that it meets the requirements to run in the system code space.

The module can be called from routines written in any language, regardless of whether the routines run in the CRE. Use this option for user code functions that do not rely on run-time libraries. It is intended for system-level programming. For additional considerations, see **System-Level Programming** on page 166.

You must verify that code does not use the C run-time library and other CRE resources. The compiler performs no verification. Specifying `ENV EMBEDDED` does not make code that uses the C run-time library and other CRE resources able to run in the system code space.

For the native C and C++ compilers, this option sets the `_EMBEDDED` feature-test macro.

`LIBRARY`

specifies that the module requires the CRE and that the module meets the requirements to run in the user library space.

The module can be called by routines written only in C or C++. Use this option for TNS user library functions. Do not use this option for native user library functions.

TNS user library programs can call a very limited number of C run-time library functions. Specifically, TNS user library programs cannot call functions that require relocatable data blocks (global data). Native user library programs can call the entire C run-time library.

For the native C and C++ compilers, this option sets the `_LIBRARY` feature-test macro.

`LIBSPACE`

specifies that the module does not require resources provided by the CRE and that it meets the requirements to run in the user library space or system library space.

You must verify that code does not use the C run-time library and other CRE resources. The compiler performs no verification. Specifying `ENV LIBSPACE` does not make code that uses the C run-time library and other CRE resources able to run in the user library space or system library space.

The module can be called from routines written in any language, regardless of whether the routines run in the CRE. It is intended for system-level programming. For additional considerations, see **System-Level Programming** on page 166.

For the native C and C++ compilers, this option sets the `_LIBSPACE` feature-test macro.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	`ENV COMMON`	`ENV COMMON`
G-series TNS `c89` utility	`ENV COMMON`	`ENV COMMON`
TNS/R native C and C++ compilers	`ENV COMMON`	`ENV COMMON`

Table Continued

	SYSTYPE GUARDIAN	SYSTYPE OSS
Native c89 and c99 utilities	ENV COMMON	ENV COMMON
TNS/E native C and C++ compilers	ENV COMMON	ENV COMMON
TNS/X native C and C++ compilers	ENV COMMON	ENV COMMON

For each of the ENV pragma options, the compiler generates different code and imposes different restrictions on which run-time library and language features can be used. **ENV Options and the Availability of Run-Time Library and Language Features** table summarizes the availability of run-time library and language features for each ENV option.

Table 28: ENV Options and the Availability of Run-Time Library and Language Features

Feature	COMMON	EMBEDDED	LIBRARY	LIBSPACE
C I/O functions	Yes	No	Yes *	No
Memory allocation functions	Yes	No	Yes	No
Main routine	Yes	No	No	No
Relocatable data blocks	Yes	Yes	No	No
Functions that set errno	Yes	No	Yes	No

* ENV LIBRARY permits the use of high-level language I/O facilities if direct access to relocatable data blocks is not needed for the operations.

Usage Guidelines

- The ENV pragma is a command-line directive that can be entered on the compiler RUN command line or be specified in the source text if it is placed before any declarations. If you are using the c89 or the c99 utility, you can specify the ENV directive with the -Wenv flag.
- Each ENV option requires certain run-time library and language features. The Binder restricts you from binding modules with incompatible ENV options. For more details, see **Restrictions on Binding Caused by the ENV Pragma** on page 343.

- Pragmas ENV EMBEDDED and ENV LIBSPACE restrict the use of C I/O functions. Guardian I/O functions are allowed.
- For TNS C and C++, pragmas ENV LIBRARY and ENV LIBSPACE restrict relocatable data blocks. Without relocatable data blocks, static data is not allowed. Therefore, the compiler allocates constants (including character string constants and data declared with a const type qualifier) in the user code space instead of the user or system data space. Because the constants are in the user code space, you cannot pass pointers to these constants between code spaces without using the CSADDR pragma. For more details, see the pragma **CSADDR** on page 218.
- The compiler does not identify the use of restricted C I/O or memory allocation run-time functions. Use of restricted functions causes a failure when the function is executed.
- The compiler identifies the use of restricted static variables.
- For native C and C++, pragmas ENV LIBRARY and ENV LIBSPACE causes the compiler to place literal constants into the read-only data area. It also causes an error to be generated for global variable declarations.
- Functions in object files listed in SEARCH pragmas retain their original ENV attributes.
- To use C run-time library functions affected by locales and the CHECK pragma under pragmas ENV EMBEDDED and ENV LIBSPACE , you must specify pragma NOCHECK and the _IGNORE_LOCALE feature-test macro in the compilation text to refer to the environmentally neutral variants of the functions.

ERRORFILE

The ERRORFILE pragma directs the compiler to send error and warning messages to a specified file location.

```
ERRORFILE { filename | define-name }
```

filename

is the name of the error-logging file. It must be a disk file name.

define-name

is the name of a MAP DEFINE that refers to a disk file. The compiler uses the disk file as the error-logging file.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	Not set	Not set
G-series TNS c89 utility	Not set	Not set
TNS/R native C and C++ compilers	Not set	Not set

Table Continued

	SYSTYPE GUARDIAN	SYSTYPE OSS
Native c89 and c99 utilities	N.A.	N.A.
TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

- The ERRORFILE pragma can appear only on the RUN command line (not in the source file) for NMC or NMCPLUS (G-series Guardian) or for CCOMP or CPPCOMP (H-series or J-series Guardian).
- The effect of the ERRORFILE pragma can be achieved with the 2> operand of the c89 or the c99 utility.
- If the specified disk file does not already exist, the compiler creates an entry-sequenced file (file code 106) using the specified file name.
- If the specified file is an existing error file with file code 106, the compiler replaces the contents of the existing file with the new error file.
- If the specified file is an existing file but is not an error file, the compiler terminates and displays the message:

  ```
  ERRORFILE not created
  ```
- The error file contains one record for each error or warning message that the compiler found during compilation. Each error record contains:
  - The fully qualified, local file name of the file in which the compiler found the error or warning
  - If you specified a DEFINE name for the name of the source file, the name of the file to which the DEFINE evaluated appears in this entry
  - The EDIT line number in which the error or warning occurred
  - The column number in the line where the compiler detected the error/warning
  - The error or warning message text

ERRORS

The ERRORS pragma directs the compiler to terminate compilation if it detects more than a specified number of errors.

```
ERRORS max-errors
```

max-errors

specifies the maximum number of errors that the compiler is to generate before terminating the compilation.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	Not set for C N.A. for C++	Not set for C N.A. for C++
G-series TNS `c89` utility	Not set for C N.A. for C++	Not set for C N.A. for C++
TNS/R native C and C++ compilers	`ERRORS 100`	`ERRORS 100`

Native
`c89`
and
`c99`
utilities
	`ERRORS 100`	`ERRORS 100`

TNS/E native C and C++ compilers	`ERRORS 100`	`ERRORS 100`
TNS/X native C and C++ compilers	`ERRORS 100`	`ERRORS 100`

Usage Guidelines

- The `ERRORS` pragma can be entered on the compiler RUN command line (not in the source file) or be specified with the `-Werrors` flag of the `c89` or the `c99` utility.
- If you do not use the `ERRORS` pragma, the TNS C compiler completes a compilation regardless of the number of errors it diagnoses.

EXCEPTION_SAFE_SETJMP

The EXCEPTION_SAFE_SETJMP pragma directs the compiler to do special processing for calls to Standard C API setjmp. This enables `setjmp/longjmp` to be used with C++ exception handling. This option is valid only for TNS/R C/C++.

The pragma default settings are

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set	Not set
Native `c89` utility	Not set	Not set
`c99` utility	N.A.	N.A.
TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	N.A.	N.A.

Usage Guidelines

- The `EXCEPTION_SAFE_SETJMP` pragma can be entered on the RUN command line (NMC, NMCPLUS) or be specified as the `-Wexception_safe_setjmp` flag of the `c89` utility.
- This pragma is valid only for TNS/R target C/C++ compilations.
- This pragma is ignored if any of ENV LIBRARY, ENV LIBSPACE, ENV EMBEDDED pragmas are specified.

EXTENSIONS

The `EXTENSIONS` pragma allows source code to use syntax extensions and standard library extensions defined by HPE.

```
[NO]EXTENSIONS
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	NOEXTENSIONS	NOEXTENSIONS
Native `c89` and `c99` utilities	NOEXTENSIONS	NOEXTENSIONS
TNS/E native C and C++ compilers	NOEXTENSIONS	NOEXTENSIONS
TNS/X native C and C++ compilers	NOEXTENSIONS	NOEXTENSIONS

Usage Guidelines

- The `EXTENSIONS` pragma can be entered on the compiler RUN command line or be specified with the `-W[no]extensions` flag of the `c89` or the `c99` utility.
- The `EXTENSIONS` pragma adds these reserved keywords:

`_alias`	**`_arg_present`**	**`_atomic_get`**
`_atomic_put`	`_baddr`	`_bitlength`
`_c`	`_callable`	`_cobol`
`_cspace`	`_extensible`	`_far`
`_fortran`	`_interrupt`	`_lowmem`
`_lowmem256`	`_lowmem64`	`_near`
`_optional`	`_pascal`	`_priv`
`_procaddr`	`_resident`	`_sg`
`_sgx`	`_tal`	`_unspecified`
`_variable`	`_waddr`	

- The `EXTENSIONS` pragma extends the grammar for the ISO/ANSI C standard.
- The `EXTENSIONS` pragma allows the #include directive to specify section names and nolist.
- The EXTENSIONS pragma defines the feature-test macro _TANDEM_SOURCE.

- The EXTENSIONS pragma changes the severity of some errors to warnings.
- Standard library extensions can be enabled by defining the feature-macro _TANDEM_SOURCE without specifying the `EXTENSIONS` pragma.
- The native mode C++ compiler expects ANSI-compliant code unless you include the EXTENSIONS pragma. To avoid warnings or errors on features that are not strictly ANSI-compliant, you must use the EXTENSIONS pragma on the command line.
- No warning is displayed when the `EXTENSIONS` pragma is used with the TNS/E and TNS/X native mode compiler and all these are true:
  - The formal parameter type is a pointer to an integer type
  - The actual parameter type is a pointer to an integer type
  - Both integer types have the same size

    If the integer types have the same size, but are not both signed or both unsigned, a warning is displayed. If the integers are of different sizes, an error occurs.

- These extensions to the C language are accepted when you specify the EXTENSIONS pragma. (Unless stated otherwise, when each of these features is compiled without extensions enabled, the behavior is for the compiler to issue a warning; compiling with extensions enabled issues no diagnostic. Generally, these C extensions are also part of the C++ proposed standard; therefore, unless there is a statement to the contrary, these features are accepted when compiling in C++ mode regardless of the setting of extensions.)
  - A translation unit (input file) can contain no declarations.
  - Bit fields might have base types that are `enums` or integral types besides int and unsigned int.
  - The last member of a struct can have an incomplete array type. It must not be the only member of the struct (otherwise, the struct would have zero size). (This is not allowed in C++.)
  - A file-scope array can have an incomplete struct, union, or enum type as its element type. The type must be completed before the array is subscripted (if it is), and by the end of the compilation if the array is not extern. (This is not allowed in C++.)
  - Static functions can be declared in function and block scopes. Their declarations are moved to the file scope. (This is not allowed in C++.)
  - enum tags can be incomplete; one can define the tag name and resolve it (by specifying the brace-enclosed list) later. (C and C++ behavior is the same.)
  - The values of enumeration constants can be given by expressions that evaluate to unsigned quantities that fit in the unsigned int range but not in the int range. (C and C++ behavior is the same.)
  - An extra comma is allowed at the end of an enum list. A remark is issued. (C and C++ behavior is the same.)
  - In an initializer, a pointer constant value can be cast to an integral type if the integral type is large enough to contain it.
  - long float is accepted as a synonym for double. (C and C++ behavior is the same.)
  - The #assert preprocessing extensions of AT&T System V release 4 are allowed. These allow definition and testing of predicate names. Such names are in a namespace distinct from all other names, including macro names.

    A predicate name is given a definition by a preprocessing directive of the form:

```
#assert name #assert name (token-sequence)
```

which defines the predicate name. In the first form, the predicate is not given a value. In the second form, it is given the value token-sequence. Such a predicate can be tested in an #if expression:

```
#name (token-sequence)
```

which has the value 1 if a #assert of that name with that token-sequence has appeared, and 0 otherwise. A predicate can be given more than one value at a given time.

A predicate can be deleted by a preprocessing directive of the form:

```
#unassert name
#unassert name (token-sequence)
```

The first form removes all definitions of the indicated prediate name; the second form removes just the indicated definition, leaving any others there might be.

- External entities declared in other scopes are visible. A warning is issued when compiled with extensions. When compiled without extensions, an error is issued. (This is not allowed in C++.)

```
void  f1(void) {
 extern void f();
}
void f2() {
 f(); /* Using out of scope declaration */
}
```

EXTERN_DATA

The `EXTERN_DATA` pragma specifies whether external data references (object declared `extern`) can use GP-relative addressing.

```
EXTERN_DATA value [variable-name [,variable-name]]

value:
    { GP_OK | NO_GP }
```

GP_OK

means that GP-relative addressing is allowed. This value is not valid for the TNS/E or TNS/X compilers.

NO_GP

means that GP-relative addressing must not be used. This is the value that typically indicates that the data is exported by an SRL.

variable-name

identifies a variable that has previously been declared.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	EXTERN_DATA NO_GP	EXTERN_DATA NO_GP
Native `c89` utility	EXTERN_DATA NO_GP	EXTERN_DATA NO_GP
`c99` utility	N.A.	N.A.
TNS/E native C and C++ compilers	EXTERN_DATA NO_GP	EXTERN_DATA NO_GP
TNS/X native C and C++ compilers	EXTERN_DATA NO_GP	EXTERN_DATA NO_GP

Usage Guidelines

- The `EXTERN_DATA` pragma can be entered on the compiler RUN command line or in the source text. It can also be specified with the `–Wextern_data` flag of the `c89` utility.
- If no *variable-name* is given, the pragma sets the global default, which remains in effect until the next EXTERN_DATA pragma. This form of the pragma can appear in source code, in the compiler RUN command line, or in a `c89` flag.
- If a *variable-name* is given, the pragma sets the addressing type to use for only the specified variable. This form of the pragma can appear only in the source. It must occur after the declaration of *variable-name*.
- `EXTERN_DATA GP_OK` is not compatible with generation of PIC (Position-Independent Code) code. The TNS/R native compiler issues a warning if this pragma is combined with the **SHARED** on page 300 pragma or **CALL_SHARED** on page 210, and ignores the `GP_OK` directive. The TNS/E and TNS/X compilers ignore this directive.
- Specifying `EXTERN_DATA GP_OK` as the global default might prevent some ISO/ANSI C Standard conforming programs from linking successfully. This might occur if the user program uses SRL data but does not use the correct header file to declare the SRL data.

  For example, the C run-time library is in an SRL, and all the data it exports (such as `errno`) is SRL data. GP-addressing cannot be used to access SRL data. The `nld` utility issues an error when an SRL data item is addressed through GP. You can correct this problem by doing one of these:

- Use the default value for `EXTERN_DATA`, which is `NO_GP`.
- Use the correct header file to declare the SRL data.
- Add a `#pragma extern_data no_gp` *SRL-variable-name* for each SRL data object that the program explicitly declares.

`EXTERN_DATA` has no effect if the compilation specifies the `SRL` directive.

Examples

```
/* “excerpt” from <errno.h> */
#pragma push extern_data
#pragma extern_data no_gp

extern int errno;  /* References to errno will not use
                      GP-relative addressing. */

#pragma pop extern_data

/* alternative form */
extern int errno;
#pragma extern_data no_gp errno
```

FIELDALIGN

The `FIELDALIGN` pragma controls the component layout of structures for compatibility between TNS and native structure layout and for compatibility with native mixed-language structure layout.

```
FIELDALIGN align-attribute [ tag-list ]

align-attribute:

    { AUTO | CSHARED2 | SHARED2 | SHARED8 | PLATFORM }

tag-list:

    tag-name [ tag-list ]
```

align-attribute

specifies the alignment rules for the compiler to follow. Valid values are:

`AUTO`

directs the compiler to choose the layout scheme. The compiler might add implicit (anonymous) fillers to ensure that components are properly aligned. This is the default alignment. The TNS C compiler and the native C and C++ compilers have different `AUTO` alignment rules.

`CSHARED2`

directs the TNS C, native C, and native C++ compilers to lay out components using the TNS C compiler `AUTO` alignment rules. Substructures and derived classes with `AUTO` alignment and pointers with platform-dependent sizes are not allowed when `CSHARED2` is specified.

SHARED2

directs the TNS C, native C, and native C++ compilers to lay out components using the default TAL compiler SHARED2 alignment rules.

SHARED8

directs the TNS C, native C, and native C++ compilers to lay out components using the pTAL compiler SHARED8 alignment rules.

PLATFORM

directs the TNS C compiler to lay out components using the SHARED2 alignment rules, and directs the native C and C++ compilers to lay out components using the AUTO alignment rules.

tag-name

specifies the name of a structure to which the given alignment rules apply. If no tag is specified, the FIELDALIGN pragma applies the structure allocation scheme to all structures in the compilation unit.

The pragma default settings are:

	SYSTYPE GUARDIAN	**SYSTYPE OSS**
TNS C compiler	FIELDALIGN AUTO	FIELDALIGN AUTO
G-series TNS c89 utility	FIELDALIGN AUTO	FIELDALIGN AUTO
TNS/R native C and C++ compilers	FIELDALIGN AUTO	FIELDALIGN AUTO
Native c89 and c99 utilities	FIELDALIGN AUTO	FIELDALIGN AUTO
TNS/E native C and C++ compilers	FIELDALIGN AUTO	FIELDALIGN AUTO
TNS/X native C and C++ compilers	FIELDALIGN AUTO	FIELDALIGN AUTO

While the TNS compilers and the native compilers default to AUTO, the AUTO alignment rules are not the same for the TNS, TNS/R, TNS/E, and TNS/X native compilers.

Usage Guidelines

- The FIELDALIGN pragma can be entered on the compiler RUN command line or in the source text. It can also be specified with the -Wfieldalign flag of the c89 or the c99 utility.
- Components whose sizes are different in TNS mode and native mode (_baddr, _waddr, _near, _sg pointers) are not allowed in CSHARED2, SHARED2, and SHARED8 structures.
- Components with AUTO alignment are not allowed in CSHARED2, SHARED2, PLATFORM, and SHARED8 structures. An error message is issued for invalid components.
- The use of pragma FIELDALIGN without a *tag-name* specifies the default for the entire compilation. This form of the pragma can be specified on the command line or at the start of the source text before anything except comments or other pragmas.
- Multiple occurrences of a FIELDALIGN pragma without a *tag-name* within a compilation is an error.
- Pragma FIELDALIGN with a *tag-name* applies only to a struct whose *tag-name* is declared in the same scope as the pragma. C has only file (global) and block (local) scope. C++ has class/struct scope in addition to file and local scopes. The pragma must occur within the same scope (and before) the struct declaration.
- Multiple occurrences of a FIELDALIGN pragma for the same *tag-name* is an error for TNS compilers but is allowed for native compilers.
- Use AUTO whenever the data layout is target independent and not shared.
- Use SHARED2 to share data between TNS programs and native programs.
- If you use the sizeof() function with a SHARED2 substruct, the actual size allocated for a struct is returned, including any trailing filler. The size of a SHARED2 substruct that has a *tag-name* is always a multiple of its alignment. Only SHARED2 substructs that do not have a *tag-name can have an odd-byte size*.
- Use PLATFORM to share data between native C/C++ and pTAL programs that run on the same platform.
- SHARED8 requires that you explicitly declare any filler needed to properly align fields. The compiler issues a warning for improperly aligned fields in a SHARED8 structure.
- For more details on structure alignment, see the *pTAL Conversion Guide* and *pTAL Reference Manual*.
- If a template class has a non-template base class or a non-template member class with the FIELDALIGN AUTO attribute, all the instantiations of the template class should have the same FIELDALIGN attribute as the non-template base or member class; otherwise the compilation of such instantiations will fail.
- For TNS C++, there are additional restrictions:
  - FIELDALIGN must be AUTO for inherited classes and their base classes and classes with virtual member functions.
  - Templates cannot be specified in FIELDALIGN pragmas and always have FIELDALIGN AUTO. Nested structs are not implemented for templates. Templates can contain non-AUTO structs declared using tags.
  - No other pragmas can appear on the same source line as #pragma FIELDALIGN.

Examples

1. Global pragma FIELDALIGN is allowed anywhere in the source, not just at the beginning, and it can also be used as an argument to pragmas PUSH and POP. For example:

```
#pragma fieldalign auto
// the global fieldalign is now set to auto

struct AutoStruct {};
#pragma push fieldalign
//save the current fieldalign value

#pragma fieldalign shared2
// the global fieldalign is now shared2In t

struct Shared2Struct{};
#pragma pop fieldalign
//global fieldalign is restored to auto
```

2. In this example, the member str1.str2.c2 starts at a byte address instead of the typical 16-bit word address:

```
#pragma fieldalign shared2 STAG
  struct STAG { char c1;
    struct { char c2;
             int i;
           } str2; /* substructure declared
                  inline without tag */
} str1;
```

3. In this example, pragma FIELDALIGN SHARED2 causes all structures in the compilation unit to be the same as TAL's definition structures, except for any structure with the tag CTAG, whose members always are 16-bit word-aligned.

```
#pragma fieldalign shared2
#pragma fieldalign auto CTAG
...
struct CTAG { ...
} cstruct;
...
```

4. This example shows the alignment of a structure with bit-field members, regardless of whether pragma WIDE is present or not:

```
#pragma fieldalign shared2 ATAG

struct ATAG { char c1;
int b1:3;
int b2:3;
int b3:12;
char c2; } s1;

s1:  ----------------------------------
    |    c1          |   filler       |
     ----------------------------------
    |  <0:2=b1><3:5=b2><6:15=filler>  |
     ----------------------------------
    |    <0:11=b3><12:15=filler>      |
     ----------------------------------
    |    c2          |  |
     ----------------------------------
```

5. This example shows the alignment of a structure with an array member:

```
#pragma fieldalign shared2

struct { char c1;
         struct { char c2;
       int i1;
       } a[2];
          } s1;

s1:   ----------------------------------
    |    c1          |     a[0].c2    |
     ----------------------------------
    |a[0].i1   |
     ----------------------------------
    |filler for a[0] |    a[1].c2      |
     ----------------------------------
    |a[1].i1   |
     ----------------------------------
    |filler for a[1] |   |
     ----------------------------------
```

FORCE_VTBL

The `FORCE_VTBL` command-line option forces definition of virtual function tables in cases where the heuristic used by the compiler to decide on definition of virtual function tables provides no guidance. The default behavior is to make the definition a local static entity. This option is valid only for native C++.

```
FORCE_VTBL
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set	Not set

Table Continued

	SYSTYPE GUARDIAN	SYSTYPE OSS
Native c89 and c99 utilities	Not set	Not set
TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

- FORCE_VTBL can be entered on the compiler RUN command line (NMCPLUS or CPPCOMP) or be specified with the -Wforce_vtbl option of the c89 or c99 utility.
- The FORCE_VTBL command-line option forces the definition of the virtual function table for classes in compilation units that do not contain the first non-inline, non-pure virtual function of the class. The FORCE_VTBL option differs from the default behavior in that it does not force the definition to be local.
- See also the description of the **SUPPRESS_VTBL** on page 314 command-line option.

FORCE_STATIC_TYPEINFO

The FORCE_STATIC_TYPEINFO command-line option forces the typeinfo variables to be static to the file. This option is applicable only to variables that are not part of an exported or imported class.

```
FORCE_STATIC_TYPEINFO
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS c89 utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set	Not set

Table Continued

	SYSTYPE GUARDIAN	SYSTYPE OSS
Native c89 and c99 utilities	Not set	Not set
TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	N.A.	N.A.

Usage Guidelines

FORCE_STATIC_TYPEINFO can be entered on the compiler RUN command line (NMCPLUS) or be specified with the `–Wforce_static_typeinfo` flag of the `c89` or the `c99` utility.

FORCE_STATIC_VTBL

The `FORCE_STATIC_VTBL` command-line option forces the virtual function tables that are created by the compiler to be static to the file and are not exported. This option is applicable only to variables that are not part of an exported or imported class.

```
FORCE_STATIC_VTBL
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set	Not set
Native `c89` and `c99` utilities	Not set	Not set
TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	N.A.	N.A.

Usage Guidelines

FORCE_STATIC_VTBL can be entered on the compiler RUN command line (NMCPLUS) or be specified with the `-Wforce_static_vtbl` flag of the `c89` or the `c99` utility.

FUNCTION

The `FUNCTION` pragma declares attributes of external routines. The `FUNCTION` pragma is used at function declaration, not at function definition.

```
FUNCTION c-function-name

   ( attribute-specifier [ , attribute-specifier ] )

attribute-specifier:
```

c-function-name

is the name used inside the program to refer to the external routine.

language

is the name of the language of the external routine. `tal` identifies both TAL and pTAL routines.

attribute

specifies a function attribute, which is one of:

`alias` **("*external-name*")**

identifies the name of the external routine used by Binder or a linker. *external-name* can include any character that Binder or a linker recognizes.

The *external-name* specification is treated:

- The TNS/R C/C++ compiler does not modify the string supplied as the *external-name* specification in any manner; that is, it does not upshift it.
- The TNS C/C++ compiler upshifts the *external-name* string if the language is cobol, fortran, pascal, or tal. However, if the language is C or unspecified, the TNS compiler does not modify the name.

The *external-name* argument must be enclosed in parentheses and quotation marks, as indicated in the syntax. For example:

```
alias ("MyTALFunction")
```

Use `alias` to describe the name of a function written in COBOL, FORTRAN, D-series Pascal, or TAL that does not have a valid C name.

`resident`

causes the function code to remain in main memory for the duration of program execution. The operating system does not swap pages of this code. Binder or a linker allocate storage for resident functions as the first functions in the code space.

`variable`

directs the compiler to treat all formal parameters of the function as though they were optional, even if some parameters are required by your code. If you add new parameters to a variable function, all callers to the function must be recompiled.

Variable functions are equivalent to VARIABLE procedures in TAL, pTAL, and D-series Pascal. For more details, see **Writing Variable and Extensible Functions** on page 168.

directs the compiler to treat all parameters of the function as though they were optional, even if some parameters are required by your code. You can add new formal parameters to an extensible function without recompiling callers unless the callers use the new parameters.

extensible functions are equivalent to EXTENSIBLE procedures in TAL, pTAL, and D-series Pascal. For more details, see **Writing Variable and Extensible Functions** on page 168.

Effects of FUNCTION Attributes of shows the effect of FUNCTION attributes for each language. “Valid” indicates that the compiler accepts the attribute. “Ignore” indicates that the compiler accepts but ignores the attribute. “Error” indicates that the compiler issues an error.

Table 29: Effects of FUNCTION Attributes

Language	No attribute	extensible	resident	variable	alias
`c`	Valid	Valid	Valid	Valid	Valid
`cobol`	Valid	Error	Ignore	Error	Valid
`fortran`	Valid	Error	Ignore	Error	Valid
`pascal`	Valid	Valid	Ignore	Valid	Valid
`tal`	Valid	Valid	Valid	Valid	Valid
`unspecified`	Valid	Valid	Ignore	Valid	Valid

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	Not set	Not set
G-series TNS `c89` utility	Not set	Not set
TNS/R native C and C++ compilers	Not set	Not set
Native `c89` and `c99` utilities	N.A.	N.A.
TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

- The FUNCTION pragma can be entered on the compiler RUN command line or in the source text. For TNS c89, the FUNCTION pragma must be entered in the source file.
- The alias attribute cannot be used with a pointer to a function.
- The extensible and variable attributes cannot be specified for a function that passes or returns a structure by value.
- Only one language can be specified for a function.
- FUNCTION can be entered in either uppercase or lowercase.
- The extensible and variable attributes cannot both be specified for the same function.
- If you declare an external procedure as unspecified, the actual procedure cannot be both written in C and compiled using the OLDCALLS pragma.
- Unlike previous methods for declaring external functions, the FUNCTION pragma is compliant with the ISO/ANSI C standard.
- The language attribute tal identifies both TAL and pTAL procedures.
- The *param-count* argument for the extensible attribute can be used only for native C and C++ programs.
- Cfront imposes these additional restrictions on the FUNCTION pragma:
  - It cannot be used for C++ functions or to change the language attribute.
  - It can be applied only to functions in the global scope.
  - No other pragmas can appear on the same line as a FUNCTION pragma.
  - The resident and alias attributes cannot be used.
- C-style variable argument lists (...) cannot be used in variable and extensible functions.
- Extensible and variable functions in native C and C++ can have a maximum of 31 parameters.
- Native C++ imposes these additional restrictions on the FUNCTION pragma:
  - Extensible and variable functions cannot be overloaded.
  - Extensible and variable functions cannot have default parameters.
  - Extensible and variable functions cannot be used in function templates and member functions.

Examples

This example shows the declaration for the C function myproc, which is a TAL variable procedure with the externally visible procedure name MY^PROC:

```
void myproc (int);
#pragma function myproc (tal, variable, alias("MY^PROC"))
/* function declaration */

void myproc (int x)
```

```
/* function definition */
{ if ( x >= 0 )
      Positive(x);

  else
      Negative(x);
}
```

GLOBALIZED

The GLOBALIZED option directs the TNS/E native C and C++ compilers to generate preemptable object code. Preemptable object code allows named references in a DLL to resolve to externally-defined code and data items instead of to the DLL's own internally-defined code and data items. You must specify the GLOBALIZED option when building globalized DLLs or when compiling code that will be linked into a globalized DLL. Preemptable code is less efficient than non-preemptable code; therefore, the GLOBALIZED option should be specified only when required. By default, the compiler generates non-preemptable object code.

```
GLOBALIZED
```

The default settings for the GLOBALIZED option are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	N.A.	N.A.
Native `c89` and `c99` utilities	Not set	Not set
TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Considerations

- The GLOBALIZED option can be specified only on the compiler RUN command line. In the OSS and PC environments, it is specified with the `–Wglobalized` flag of the `c89` or the `c99`utility.
- In the OSS and Windows environments, the `–Wglobalized` option has an effect only for `–Wtarget=ipf` or `-Wtarget=tns/e`. It has no effect (and no diagnostic is issued) for `-Wtarget=mips` or `–Wtarget=tns/r` or `tns/x` or `x86`.
- Non-preemptable code is more efficient that preemptable code, and results in faster compilation and execution. You should specify the GLOBALIZED option only when building globalized DLLs or compiling code that will be linked into a globalized DLL.
- If the linker is invoked by the compiler, the linker option

  ```
  -b globalized
  ```

is automatically specified only when a DLL is being built by the linker.

HEADERS

The HEADERS pragma directs the native C and C++ compilers to print a list of included header files.

```
HEADERS
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS c89 utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set	Not set
Native c89 and c99 utilities	Not set	Not set
TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

- The HEADERS pragma must be entered on the compiler RUN command line, not in the source text.
- This pragma causes the compiler to act like a limited preprocessor. No compilation is performed.
- If both pragmas CPPONLY and HEADERS are specified, CPPONLY takes precedence.

HEAP

The HEAP pragma specifies the maximum heap size of a program compiled with the RUNNABLE pragma.

```
HEAP count [ { BYTES | PAGES | WORDS } ]
```

count [{ BYTES | PAGES | WORDS }]

specifies the maximum size of the heap in terms of a number of units of a given size. *count* specifies the number of units, and the keyword following *count* specifies the size of each unit. If you omit a keyword, the compiler assumes BYTES.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	`HEAP 1 PAGES`	`HEAP 1 PAGES`
G-series TNS `c89` utility	`HEAP 1 PAGES`	`HEAP 1 PAGES`
TNS/R native C and C++ compilers	Not set (see **Usage Guidelines** on page 244)	Not set (see **Usage Guidelines** on page 244)
Native `c89` and `c99` utilities	Not set (see **Usage Guidelines** on page 244)	Not set (see **Usage Guidelines** on page 244)
TNS/E native C and C++ compilers	Not set (see **Usage Guidelines** on page 244)	Not set (see **Usage Guidelines** on page 244)
TNS/X native C and C++ compilers	Not set (see **Usage Guidelines** on page 244)	Not set (see **Usage Guidelines** on page 244)

Usage Guidelines

- If the `RUNNABLE` pragma is not specified, the `HEAP` pragma does not affect the compilation.
- One page equals 2048 bytes (2 kilobytes).
- If you do not use the `HEAP` pragma when compiling a TNS program with the `RUNNABLE` pragma, the compiler automatically provides a heap of one page.
- For native C/C++, the `HEAP` pragma must be entered on the compiler RUN command line; for TNS C and `c89` in the OSS environment, the pragma can be entered either on the compiler RUN command line or in the source text.
- Native C and C++ programs use a different heap architecture than TNS programs. Because of this, native C and C++ programs normally do not need to specify maximum heap size; the default value for the `heap_max` attribute of a program file is zero, indicating that the size of the heap is limited only by available resources.
- For native C and C++ programs, you should not specify the HEAP pragma unless there is an unusual reason to limit the heap size to less than the capacity of the program address space.

  The `HEAP` pragma specifies an upper bound on heap size by setting the `heap_max` attribute in a new loadfile; for example, specifying `HEAP` *n* `BYTES` is equivalent to specifying `-set heap_max` *n* to the `eld` linker for the loadfile.

  You do not need to recompile to change the setting. The `HEAP` pragma setting and `-set heap_max` setting can be replaced later by using the linker command `-change heap_max` *n filename* , where *filename* designates the program file. See the appropriate linker manual for information on setting the `heap_max` flag with the appropriate number of bytes.

- If you do not use the HEAP pragma when compiling a native program with the RUNNABLE pragma, the compiler leaves the maximum heap size unspecified. This causes the value used to be determined by the first of these that has a nonzero value:

  1. The pe_heap_max value assigned to the program file by a process that launches it
  2. The heap_max attribute of the program file

  If the heap_max attribute remains zero, all the user data area (minus that area used for global data segments and the argv[] and envp[] arrays) is available for the heap.

- If you do use the HEAP pragma when compiling a native program, the space specified must be sufficient to contain at least the global data segments and the argv[] and envp[] arrays.
- The maximum size of native user heap can be affected by the use of operating system memory segments:

  - Flat segments are allocated in the same area, downward from its upper bound.
  - Certain products allocate segments in the same area, causing address collisions when certain features are used; the *QIO Configuration and Management Manual* provides guidance on one such instance.
  - DLL text and data segments can be loaded into the same area.

    When such segments already exist, the heap area is bounded by the start of the segment with the lowest address. An attempt to allocate a flat segment or load a DLL can fail if the heap has already consumed the area needed.

- Heap growth, like the allocation or growth of other data segments, will fail if insufficient swap space is available for the processor.

  If your program causes a run-time error message that it is out of virtual memory, try increasing the swap file space available to it, following the guidelines in the *Kernel-Managed Swap Facility (KMSF) Manual*.

HIGHPIN

The HIGHPIN pragma specifies that the object file should be run at a high PIN (256 or greater) or at a low PIN (0 through 254).

```
[NO]HIGHPIN
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	Not set	Not set
G-series TNS c89 utility	HIGHPIN	HIGHPIN
TNS/R native C and C++ compilers	HIGHPIN	HIGHPIN

Table Continued

	SYSTYPE GUARDIAN	SYSTYPE OSS
Native `c89` and `c99` utilities	`HIGHPIN`	`HIGHPIN`
TNS/E native C and C++ compilers	`HIGHPIN`	`HIGHPIN`
TNS/X native C and C++ compilers	`HIGHPIN`	`HIGHPIN`

Usage Guidelines

- For the TNS C compiler, Cfront, and the TNS `c89` utility, the `HIGHPIN` pragma can be placed in the source text or in the RUN command that executes the compiler.

  When you bind several object files together, Binder sets the HIGHPIN attribute in the target object file if, and only if, all the object files being bound together have the `HIGHPIN` pragma specified.

- For the native C and C++ compilers, the `HIGHPIN` pragma can be entered on the compiler RUN command line (not in the source file) or be specified with the `-Whighpin` flag of the `c89` or the `c99` utility.

  The HIGHPIN attribute is set for native C and C++ programs only if an executable object file is the output of the compilation. (Process attributes cannot be set for native relinkable object files.)

- When the operating system creates a process, it assigns a process identification number (PIN) to the process. Systems running D-series RVUs or later support these ranges of PINs:

Low-PIN range	0 through 254
High-PIN range	256 through the maximum number supported for the processor

- To run an object file at high PIN from the TACL prompt, these conditions must be met:

  - Your processor is configured for more than 256 process control blocks (PCBs).
  - High PINs are available in your processor.
  - Your object file has the HIGHPIN attribute set.
  - The TACL HIGHPIN built-in variable or the HIGHPIN run-time parameter is set.

  If these four conditions are met, the operating system assigns a high PIN, if available. If no high PINs are available, the operating system assigns a low PIN.

  You can set the HIGHPIN flag of TNS and accelerated object files either:

  - During compilation by using the `HIGHPIN` pragma
  - After compilation using a Binder command

  You can set the HIGHPIN flag of a native object file either:

- During compilation by using the HIGHPIN pragma, if an executable file is produced by the compilation
- After compilation and linking of an executable object file by using an eld, xld, ld, or nld command

- For more details on using high PINs and converting a C program to run at a high PIN, see the *Guardian Programmer's Guide*.

HIGHREQUESTERS

The HIGHREQUESTERS pragma specifies that the object file supports high PIN requesters if the object file includes the main function.

```
[NO]HIGHREQUESTERS
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	Not set	Not set
G-series TNS c89 utility	HIGHREQUESTERS	HIGHREQUESTERS
TNS/R native C and C++ compilers	HIGHREQUESTERS	HIGHREQUESTERS
Native c89 and c99 utilities	HIGHREQUESTERS	HIGHREQUESTERS
TNS/E native C and C++ compilers	HIGHREQUESTERS	HIGHREQUESTERS
TNS/X native C and C++ compilers	HIGHREQUESTERS	HIGHREQUESTERS

Usage Guidelines

- For the TNS C compiler, Cfront, and the TNS c89 utility, the HIGHREQUESTERS pragma can be placed in the source text or in the RUN command that executes the compiler.

  When you bind several object files together, Binder sets the HIGHREQUESTERS attribute in the target object file if, and only if, the object file with the main function has the HIGHREQUESTERS attribute.

- For the native C and C++ compilers, the HIGHREQUESTERS pragma can be entered on the compiler RUN command line (not in the source file) or be specified with the -Whighrequesters flag of the c89 or the c99 utility.

  The HIGHREQUESTERS attribute is set for native C and C++ programs only if an executable object file is the output of the compilation. (Process attributes cannot be set for native relinkable object files.)

- You can set the HIGHREQUESTERS object-file attribute either during compilation using the HIGHREQUESTERS pragma or after compilation using the Binder SET command (for TNS programs) or the eld, xld, ld, or nld utility (for native programs).
- For more details on setting the HIGHREQUESTERS object-file attribute, see the *Guardian Programmer's Guide*.

ICODE

The ICODE pragma controls whether the compiler listing includes the instruction-code mnemonics generated for each function immediately following the source text of the function. The ICODE pragma directs the compiler to list these mnemonics, and NOICODE directs it to not list them.

```
[NO]ICODE
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	NOICODE	NOICODE
G-series TNS c89 utility	NOICODE	NOICODE
TNS/R native C and C++ compilers	N.A.	N.A.
Native c89 and c99 utilities	N.A.	N.A.
TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	N.A.	N.A.

Usage Guidelines

- The instruction code listing does not show final G-plus addresses for global variables. Instead, the addresses are relative to the start of the block. G-plus addresses for blocks are determined by the bind-time layout. To display the final addresses, use Binder or Inspect commands. The instruction-code listing does not show the procedure entry-point (PEP) entry in PCAL instructions. You can get this information from the bind map.
- The native C and C++ compilers do not support these pragmas. Use the INNERLIST pragma instead.

IEEE_FLOAT

The IEEE_FLOAT directive specifies that the native C and C++ compilers use the IEEE floating-point format for performing floating-point operations. Compare to pragma **TANDEM_FLOAT** on page 319.

```
IEEE_FLOAT
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS c89 utility	N.A.	N.A.
TNS/R native C and C++ compilers	TANDEM_FLOAT	TANDEM_FLOAT
Native c89 and c99 utilities	TNS/R code: TANDEM_FLOAT TNS/E code: IEEE_FLOAT TNS/X code: IEEE_FLOAT	TNS/R code: TANDEM_FLOAT TNS/E code: IEEE_FLOAT TNS/X code: IEEE_FLOAT
TNS/E native C and C++ compilers	IEEE_FLOAT	IEEE_FLOAT
TNS/X native C and C++ compilers	IEEE_FLOAT	IEEE_FLOAT

Usage Guidelines

- The IEEE_FLOAT directive can be entered on the compiler RUN command line or, in the OSS environment, with the -WIEEE_float flag of the c89 or the c99 utility.
- To build and run programs using IEEE floating-point format, you need these software and hardware:
  - NonStop server with an Itanium processor or NonStop S7000 or S72000 server (IEEE floating-point format is not available on NonStop S70000 processors)
  - NonStop OS G06.06 or later product version
  - G06.06 or later product version of the native C or C++ compiler
  - G06.06 or later product version of the eld, xld, ld, or nld utility for linking.
- You must also specify the VERSION2 directive if you specify IEEE_FLOAT with the TNS/R native C++ compiler. For more details, see **VERSION2** on page 324.
- If you specify the IEEE_FLOAT directive, you cannot also specify the SQL pragma.
- Note these advantages of using IEEE floating-point:
  - IEEE floating-point format has a different range of values and different precision than Tandem floating-point format.
  - IEEE floating-point format gives faster performance and is easier for porting native applications than Tandem floating-point format.
  - IEEE floating-point default handling of overflow, underflow, divide-by-zero, and invalid operation is better than the Tandem floating-point handling.
  - IEEE floating-point default rounding mode gives more accurate results.
  - IEEE floating-point directed roundings and “sticky” flags are useful debugging tools for investigating calculations that might go wrong because of rounding problems, division by zero, or other

problems. Sticky flags are exception flags that stay set until explicitly reset by the user. Sticky flags allow the user to check a chain of computations.

  - IEEE floating-point denormalized numbers avoid computational problems that arise from very small numbers as intermediate results in computations.

- The **IEEE and Tandem Floating-Point Macro Values in float.h** table lists the macros in the `float.h` file that have different values for the two floating-point formats.

Table 30: IEEE and Tandem Floating-Point Macro Values in float.h

Macro Name	Tandem Format	IEEE Format
FLT_ROUNDS	1	1
FLT_RADIX	2	2
FLT_MANT_DIG	23	24
DBL_MANT_DIG	55	53
FLT_EPSILON	2.3841858e-07	1.9209290e-07
DBL_EPSILON	5.551115123125782720e-17	2.2204460492503131e-16
FLT_MIN_10_EXP	(-77)	(-37)
DBL_MIN_10_EXP	(-77)	(-307)
FLT_MAX_10_EXP	77	38
DBL_MAX_10_EXP	77	308
FLT_MIN_EXP	(-254)	(-125)
DBL_MIN_EXP	(-254)	(-1021)
FLT_MAX_EXP	254	128
DBL_MAX_EXP	254	1024
FLT_DIG	6	6
DBL_DIG	16	15
FLT_MIN	1.7272337e-77	1.17549435E-38F
DBL_MIN	1.7272337110188889e-77	2.2250738585072014E-308
FLT_MAX	1.1579208e77	3.40282347E+38F
DBL_MAX	1.15792089237316192e77	1.7976931348623157E+308

- In addition to the differences shown in the **IEEE and Tandem Floating-Point Macro Values in float.h** table, note these:

  - For the `float` type in IEEE floating-point format, the smallest positive nonzero number is approximately 1.40129846E-45. This is a denormalized number; Tandem floating-point format does not have denormalized numbers.
  - For the `double` type in IEEE floating-point format, the smallest positive nonzero number is approximately 4.9406564584124654E-324. This is a denormalized number.
  - IEEE floating-point format has special values for Infinity and Not a Number (NaN). In addition, for every value except NaN values, there is a positive value and a negative value. This includes infinities (+Infinity and -Infinity) and zeroes (+0 and -0).

- When converting from fixed-point to floating-point format or from a floating-point number to a narrower floating-point number, IEEE floating point typically rounds to the nearest value according to the current IEEE floating-point rounding mode.

  When converting from floating-point to fixed-point formats, IEEE floating-point normally truncates the nearest representable value, as specified by the ANSI C standard.

  The IEEE floating-point standard requires a way to convert from floating point to fixed point with rounding. To make this conversion, use the `rint()` function and then cast the result to an integer type.

- A new header file (`ieeefp.h`) has been introduced which contains interfaces that apply only to IEEE floating point and are callable only when compiling using the `IEEE_FLOAT` pragma. Calling the functions in `ieeefp.h` when generating TNS floating-point code causes a syntax error because of incompatible floating-point calling conventions. If you want to use any of the resources defined by `ieeefp.h`, you must include it in your source.
- For more details about compiling and linking programs that use IEEE floating-point format, see **Compiling and Linking Floating-Point Programs** on page 369.

INLINE

For TNS C programs, the `INLINE` pragma controls whether the compiler generates inline code for certain standard library functions instead of generating a function call. `INLINE` directs the compiler to generate inline code. `NOINLINE` directs the compiler to revert to the library function call.

For native C++ and c99 programs, the `INLINE` pragma controls whether functions declared inline are actually generated inline. NOINLINE suppresses inline generation of all functions. INLINE causes the compiler to attempt to generate inline code for functions declared `inline`.

The purpose of the `INLINE` pragma is to provide more efficient code generation.

```
[NO]INLINE
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	NOINLINE	NOINLINE
G-series TNS c89 utility	NOINLINE	NOINLINE
TNS/R native C and C++ compilers	N.A. for C, INLINE for C++	N.A. for C, INLINE for C++
Native c89 and c99 utilities	N.A. for c89, INLINE for c99	N.A. for c89, INLINE for c99
TNS/E native C and C++ compilers	N.A. for C, INLINE for C++	N.A. for C, INLINE for C++
TNS/X native C and C++ compilers	N.A. for C, INLINE for C++	N.A. for C, INLINE for C++

Usage Guidelines

- For the native C++ compiler, the INLINE pragma must be entered on the compiler RUN command line. For TNS C and c89 in the OSS environment, the pragmas can be entered on the compiler RUN command line. For native c89 and c99 utilities, -W[no]inline flag is specified on the compiler RUN command line.
- For the TNS/E and TNS/X native compiler, inlining is not performed for optimization level 0.
- For TNS C programs, these functions are affected by the INLINE pragma: strlen(), strcat(), strcmp(), strcpy(), abs(), labs(), memchr(), memcmp(), memcpy(), and memset().

  When compiling with the INLINE pragma, the maximum string limit size for the strlen(), strcat(), and strcpy() functions is 65,535 bytes. When the NOINLINE pragma is in effect, the string limit size is 32,767 bytes.
- When the INLINE pragma is in effect and a program contains a locally defined function that has the same name as one of the functions that are affected by theINLINE pragma, the compiler overrides the locally defined function with the standard ANSI code. It is necessary to revert to non-inline mode to execute the locally defined function.
- This example, which applies only to TNS C and C++, shows how to mix the inlined standard definition of strlen and a local version of strlen.

```
#include <stringh>
size_t strlen(char *p)
{
   return -1;      /* locally defined, always returns -1 */
}
main()
```

```
{
   size_t i;
   #pragma inline
   i = strlen("abc");  /* i == 3 by standard ANSI
                                specification */
   #pragma noinline
   i = strlen("abc");  /* i == -1 locally defined */
}
```

INLINE_COMPILER_GENERATED_FUNCTIONS

The `INLINE_COMPILER_GENERATED_FUNCTIONS` command-line option allows all compiler-generated functions to be inlined. The default behavior is that compiler-generated functions are not inlined and are exported.

```
INLINE_COMPILER_GENERATED_FUNCTIONS
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set	Not set
Native `c89` and `c99` utilities	Not set	Not set
TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	N.A.	N.A.

Usage Guidelines

INLINE_COMPILER_GENERATED_FUNCTIONS can be entered on the compiler RUN command line (NMCPLUS) or be specified with the `-Winline_compiler_generated_functions` flag of the `c89` or the `c99` utility.

NOTE:

The INLINE_COMPILER_GENERATED_FUNCTIONS pragma affects the compiler-generated constructor, destructor, and assignment operator (operator=).

INLINE_LIMIT

The `INLINE_LIMIT` command-line option specifies the maximum number of lines that the compiler can inline. Where, *n* denotes the total number of lines in a member function. The inline limit count, *n*, is specified as an integer. If *n* is 0, then there is no limit on the number of lines that the compiler can inline. The default value is 10.

This limit applies only to member functions that are defined within their class definition without the inline specifier.

For a function not to be inlined, set `INLINE_LIMIT` to an integer lesser than the total number of the actual lines between the start position of a function definition and the end of the function. This is just information for the compiler and it is not guaranteed that the function having lesser source lines than `INLINE_LIMIT` will be inlined.

```
INLINE_LIMIT n
```

n

denotes the number of lines as an integer.

If *n* is 0, there is no limit on the number of lines that the compiler can inline. *n* should be in the range 0 through 2147483647.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set	Not set
Native `c89` utility	Not set	Not set
`c99` utility	N.A.	N.A.
TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	N.A.	N.A.

Usage Guidelines

- `INLINE_LIMIT` can only be entered on the compiler RUN command line (NMCPLUS) or be specified with the `-Winline_limit=`*n* flag of the `c89` utility.
- This pragma is only valid for TNS/R-target compilations.

INLINE_STRING_LITERALS

The `INLINE_STRING_LITERALS` command-line option allows the compiler to inline functions that take the address of a string literal.

```
INLINE_STRING_LITERALS
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS c89 utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set	Not set
Native c89 and c99 utilities	Not set	Not set
TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	N.A.	N.A.

Usage Guidelines

- INLINE_STRING_LITERALS can be entered on the compiler RUN command line (NMCPLUS) or be specified with the –Winline_string_literalsflag of the c89 or the c99 utility.
- If a function is inlined by this specification, its program will not conform to section 7.1.2 of the 1998 ISO C++ standard.

INNERLIST

The INNERLIST pragma controls whether the compiler listing includes the instruction-code mnemonics generated for each statement immediately following the source text of the statement. The INNERLIST pragma directs the compiler to list these mnemonics, and NOINNERLIST directs it not to list them.

```
[NO]INNERLIST
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	NOINNERLIST	NOINNERLIST
G-series TNS c89 utility	NOINNERLIST	NOINNERLIST
TNS/R native C and C++ compilers	NOINNERLIST	NOINNERLIST
Native c89 and c99 utilities	NOINNERLIST	NOINNERLIST
TNS/E native C and C++ compilers	NOINNERLIST	NOINNERLIST
TNS/X native C and C++ compilers	NOINNERLIST	NOINNERLIST

Usage Guideline

The INNERLIST pragma can be entered on the compiler RUN command line or in the source text. It can also be specified with the -W[no]innerlist flag of the c89 or the c99 utility.

INSPECT

The INSPECT pragma controls whether the symbolic debugger or the default system debugger is used as the default debugger for the object file. The INSPECT pragma specifies a symbolic debugger as the default debugger, and the NOINSPECT pragma specifies the system default debugger as the default debugger.

```
[NO]INSPECT
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	INSPECT	INSPECT
G-series TNS c89 utility	INSPECT	INSPECT
TNS/R native C and C++ compilers	INSPECT	INSPECT
Native c89 and c99 utilities	INSPECT	INSPECT
TNS/E native C and C++ compilers	INSPECT	INSPECT
TNS/X native C and C++ compilers	INSPECT	INSPECT

Usage Guidelines

- For native C/C++, the INSPECT pragma must be entered on the compiler RUN command line. For TNS C, the pragma can also be entered in the source file. In the OSS environment, the pragma must be specified with the –W[no]inspect flag of the c89 or the c99 utility.
- For TNS programs, the last INSPECT or NOINSPECT pragma in a translation unit determines the default debugger for the entire translation unit.
- The NOINSPECT attribute is set for native C and C++ programs only if an executable object file (loadfile) is the output of the compilation. Process attributes cannot be set for native linkfiles.
- You can set or change the INSPECT object-file attribute to ON or OFF after compilation using the Binder SET command (for TNS programs) or the eld, ld, or nld utility (for native programs).
- The INSPECT and SAVEABEND pragmas are interdependent:

    - If you specify `NOINSPECT`, the compiler automatically disables `SAVEABEND` (as though you had explicitly specified `NOSAVEABEND`).
    - If you specify `SAVEABEND`, the compiler automatically enables the symbolic debugger (as though you had explicitly specified `INSPECT`).

- The `SYMBOLS` pragma affects the `INSPECT` pragmas: if you specify `SYMBOLS` , the compiler automatically enables the symbolic debugger (as though you had explicitly specified `INSPECT`).
- To use all the features of a symbolic debugger, specify the `SYMBOLS` pragma so that symbol information is included in the object file.
- The debugging utility selected by `INSPECT` and `NOINSPECT` varies by system, environment, and available subsystem:
    - For a TNS process:
        - `NOINSPECT`

          selects, in order of precedence:

TNS/R system	TNS/E or TNS/X system
DEBUG	Inspect
	Native Inspect

        - `INSPECT`

          selects, in order of precedence:

TNS/R system	TNS/E or TNS/X system
Visual Inspect	Visual Inspect (only TNS/E)
Inspect	Inspect
DEBUG	Native Inspect

    - For a native process:
        - `NOINSPECT`

          selects, in order of precedence:

TNS/R system	TNS/E or TNS/X system
DEBUG	Native Inspect

        - `INSPECT`

          selects, in order of precedence:

TNS/R non-PIC	TNS/R PIC	TNS/E or TNS/X
Visual Inspect	Visual Inspect	Visual Inspect (only TNS/E)
Inspect	DEBUG	Native Inspect
DEBUG		

When the TACL RUNV command or OSS `runv` utility is used, then this debugger pragma selects:

- Visual Inspect on TNS/R servers
- Visual Inspect on TNS/E servers for `INSPECT`
- Native Inspect on TNS/E servers for `NOINSPECT`
- Native Inspect on TNS/X servers for `NOINSPECT`

  When the TACL RUN or RUND command or OSS `run` utility is used, then this debugger pragma selects the debugger using the lists shown above.

For more information on...	See the...
DEBUG	*DEBUG Manual*
Inspect	*Inspect Manual*
Native Inspect	*Native Inspect Manual*
Visual Inspect	VI online help on the client PC

KR

The KR pragma specifies to the native C compiler that Kernighan & Ritchie or common-usage C is being compiled instead of ISO/ANSI Standard C.

```
KR
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set for C, N.A. for C++	Not set for C, N.A. for C++

Table Continued

	SYSTYPE GUARDIAN	SYSTYPE OSS
Native c89 utility	Not set for C, N.A. for C++	Not set for C, N.A. for C++
c99 utility	N.A.	N.A.
TNS/E native C and C++ compilers	Not set for C, N.A. for C++	Not set for C, N.A. for C++
TNS/X native C and C++ compilers	Not set for C, N.A. for C++	Not set for C, N.A. for C++

Usage Guidelines

- For native C and C++, the KR pragma must be entered on the compiler RUN command line. For OSS, the KR pragma can be specified with the -Wkr flag of the c89 utility.
- The default is ISO/ANSI Standard C.
- For NonStop systems, HPE does not provide a complete set of header files that conform to Kernighan & Ritchie (K&R) C. The user must modify existing header files or obtain header files from another source to compile K&R source files.

LARGESYM

The LARGESYM pragma directs the TNS C compiler and Cfront to generate all the symbols for a given compilation to a single symbols data block containing information used by the Inspect debugger to display information about a variable or to display its contents. You must also specify the SYMBOLS pragma when you specify LARGESYM.

```
LARGESYM
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	Not set	Not set
G-series TNS c89 utility	Not set	Not set
TNS/R native C and C++ compilers	N.A.	N.A.

Table Continued

	SYSTYPE GUARDIAN	SYSTYPE OSS
Native `c89` and `c99` utilities	N.A.	N.A.
TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	N.A.	N.A.

Usage Guidelines

- The `LARGESYM` pragma can be entered on the compiler RUN command line or in the source text. For OSS, `LARGESYM` must be entered in the source file.
- Use the `LARGESYM` pragma only in a file that contains a `#include` directive for a header file that contains embedded conditional compilations that might cause a structure definition to vary from module to module. If the `LARGESYM` directive is used in all compilations, the symbol region might be extremely large. The default action for the compiler is to generate an optimized symbol region.
- The native C and C++ compilers do not support this pragma; the compilers generate complete symbols information.

Example

Assume the header file `defh` contains:

```
struct worddef {
#ifdef MACHINE_WORDLEN_32
              int intvar1;
              int intvar2;
#else
              long intvar;
#endif };
```

Further, assume that `module1c` includes the header file `defh` and that the structure `worddef` contains members `intvar1` and `intvar2`. Also assume that `module2c` includes the header file `defh` and contains a single member `intvar`. If the `LARGESYM` pragma is not used, the compiler creates a single data block that contains only one of the declarations for the structure `worddef`. When you try to display a variable of type `worddef` in the Inspect debugger, the debugger might display the wrong information.

LD(arg)

The `LD` pragma specifies arguments to be passed to the ld utility, the TNS/R linker for PIC (Position-Independent Code).

```
LD(arg)
```

arg is any argument accepted by the `ld` utility.

For more details on valid syntax and semantics, see the *ld Manual*.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set	Not set
Native `c89` utility	Not set	Not set
`c99` utility	N.A.	N.A.
TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	N.A.	N.A.

Usage Guidelines

- On Guardian environment, the `LD` pragma must be entered on the compiler RUN command line for TNS/R native C/C++. On OSS environment, specify the `LD` pragma by using the `-Wld=`*`arg`* option with the `c89` utility.
- If you are linking non-PIC files, you must use the **NLD(arg)** on page 273 pragma, which specifies arguments to the TNS/R native non-PIC linker, the nld utility.
- The `LD` pragma does not actually invoke the `ld` linker. To invoke `ld` , you must include other pragmas such as **RUNNABLE** on page 292, **SHARED** on page 300, or **LINKFILE** on page 262 . If `ld` is not invoked, this pragma is ignored.
- When you specify the `LD` pragma, you cannot also specify **NLD(arg)** on page 273 or **NON_SHARED** on page 275.
- This TNS/R native C command example shows the `LD` pragma. Note that if you specify`SHARED`, you would not need to also include `RUNNABLE`:

  ```
  nmc /in prog1/ prog1o;ld(-set highpin off -set highrequester
  off),call_shared, runnable
  ```

LINES

The `LINES` pragma specifies the maximum number of output lines per page for the compiler listing file.

```
LINES lines_per_page
```

lines_per_page

specifies the maximum number of text lines per page for the listing file. *lines_per_page* must be an integer in the range 10 through 32767.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	`LINES 60`	`LINES 60`
G-series TNS `c89` utility	`LINES 60`	`LINES 60`
TNS/R native C and C++ compilers	`LINES 60`	`LINES 60`

Native
`c89`
and
`c99`
utilities
	`LINES 66`	`LINES 66`

TNS/E native C and C++ compilers	`LINES 60`	`LINES 60`
TNS/X native C and C++ compilers	`LINES 60`	`LINES 60`

Usage Guideline

For native C and C++. the `LINES` pragma must be entered on the compiler RUN command line. For OSS, the `LINES` pragma can be specified with the `-Wlines` flag of the `c89` or the `c99` utility.

LINKFILE

The `LINKFILE` pragma invokes the appropriate linker and specifies a command file to be passed to either:

- The `nld` utility when working with conventional code
- The `eld`, `xld`, or `ld` utility when working with PIC (Position-Independent Code), as when compiling a dynamic-link library (DLL)

```
LINKFILE "file-name"
```

file-name

specifies a valid command file for the `eld` and `xld` utility, the `ld` utility, or the `nld` utility.

- For syntax and semantics of `eld` and `xld` command files, see the *`eld and xld` Manual.*
- For syntax and semantics of `ld` command files, see the *`ld` Manual* .
- For syntax and semantics of `nld` command files, see the *`nld` Manual.*

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set	Not set

Native
`c89`
and
`c99`
utilities
	N.A.	N.A.

TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

- On Guardian environment, the `LINKFILE` pragma must be entered on the compiler RUN command line for native C and C++.
- `LINKFILE` invokes `eld`, `xld`, or `ld` rather than `nld` if you include **SHARED** on page 300 or **CALL_SHARED** on page 210.
- Use the `LINKFILE` pragma to specify in a text file the names of the object files or linkfiles (not source files) that make up a program for linking with a linker utility. This behavior is applicable only for TNS/R as `ld` and `nld` are TNS/R linkers. For TNS/E,`eld` is the only linker. For TNS/X, `xld` is the only linker. An example of `LINKFILE` appears in **Examples** on page 392.
- The C or C++ run-time libraries are linked to the program only if the `RUNNABLE` pragma is also specified on Guardian environment. On OSS environment, you must specifically invoke the linker.
- The compiler driver does not verify the existence or the readability of the command file specified in the `LINKFILE` directive.

LIST

The `LIST` pragmas control the generation of compiler-listing text. The `LIST` pragma enables the generation of compiler-listing text, and the `NOLIST` pragma disables it.

```
[NO]LIST
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	LIST	LIST
G-series TNS c89 utility	LIST	LIST
TNS/R native C and C++ compilers	LIST	LIST
Native c89 and c99 utilities	LIST	LIST
TNS/E native C and C++ compilers	LIST	LIST
TNS/X native C and C++ compilers	LIST	LIST

Usage Guideline

The LIST and NOLIST pragmas can be entered on the compiler RUN command line or in the source text. You can also use the -W[no]list flag of the c89 or the c99 utility.

LMAP

The LMAP pragma controls the generation and presentation of load-map information in the compiler listing. The LMAP pragma enables load-map generation and specifies how to present the load maps. NOLMAP disables generation of load maps or disables one form of load-map presentation.

```
[NO]LMAP { lmap-type                     }
         { ( lmap-type [ , lmap-type ] ) }
         { *                             }

lmap-type:

   { ALPHA | LOC }
```

lmap-type

specifies the type of load-map presentation to enable or disable. The available presentation types are:

ALPHA

specifies load maps of functions and data blocks sorted by name.

LOC

specifies load maps of functions and data blocks sorted by starting address.

*

specifies both ALPHA and LOC.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	NOLMAP *	NOLMAP *
G-series TNS c89 utility	NOLMAP *	NOLMAP *
TNS/R native C and C++ compilers	N.A.	N.A.
Native c89 and c99 utilities	N.A.	N.A.
TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	N.A.	N.A.

Usage Guidelines

- For the TNS C compiler, the LMAP pragma can be entered on the compiler RUN command line or in the source text. For OSS, the LMAP pragma must be entered in the source file.
- The native C and C++ compilers do not support this pragma. Native compilers do not generate load map information.

MAP

The MAP pragma controls the generation of identifier maps in the compiler listing. The MAP pragma enables generation of identifier maps, and NOMAP disables it.

```
[NO]MAP
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	NOMAP	NOMAP
G-series TNS c89 utility	NOMAP	NOMAP
TNS/R native C and C++ compilers	NOMAP	NOMAP
Native c89 and c99 utilities	NOMAP	NOMAP
TNS/E native C and C++ compilers	NOMAP	NOMAP
TNS/X native C and C++ compilers	NOMAP	NOMAP

Usage Guidelines

- The `MAP` pragma can be entered on the compiler RUN command line or in the source text. It can also be specified with the `-W[no]map` flag of the `c89` or the `c99` utility.
- Identifier maps display information about local and global identifiers, and they appear at the end of the compiler listing.

MAPINCLUDE

The `MAPINCLUDE` pragma specifies how to transform file names within `#include` directives.

```
MAPINCLUDE [ FILE | PATH ] "from" = "to"
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS and PC
TNS C compiler	Not set	Not set
G-series TNS c89 utility	N.A.	N.A.

Table Continued

	SYSTYPE GUARDIAN	SYSTYPE OSS and PC
TNS/R native C and C++ compilers	Not set	Not set
Native c89 and c99 utilities	N.A.	N.A.
TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

- The MAPINCLUDE pragma can be specified only in the source file.
- PRAGMA MAPINCLUDE " *from* "="*to* " directs the compiler to replace any name in an #include directive that fully matches the *from* string with the *to* string.
- PRAGMA MAPINCLUDE PATH "*from* "="to" directs the compiler to replace any directory pathname in an #include directive that begins with the *from* string with the to string.
- PRAGMA MAPINCLUDE FILE "*from* "="to “ directs the compiler to replace any file name in an #include directive that ends with the *from* string with the to string.
- Only one MAPINCLUDE pragma without PATH or FILE can be performed on each #include.
- A MAPINCLUDE FILE and a MAPINCLUDE PATH can both operate on the same -#include.
- Only one MAPINCLUDE pragma can be specified for each *from* string.
- For additional guidelines and examples of using the MAPINCLUDE pragma with class libraries such as Tools.h++, see **Pragmas for Tools.h++** on page 108 .
- These two examples illustrate the significance of specifying or omitting the `FILE` option with the MAPINCLUDE pragma. You have these two `#include` directives:

  ```
  #include "machin.h"
  #include "n.h"
  ```

  - This `MAPINCLUDE` pragma changes only the exact file name `n.h` and does not affect the file name `machin.h`:

    ```
    mapinclude "n.h" = "e95nh"
    ```

  - This `MAPINCLUDE` pragma with the `FILE` option changes the file name `machin.h` to `machie9h` after file-name compression. This is because the original file name ends in `n.h`:

    ```
    MAPINCLUDE FILE n.h = e95nh
    ```

However, the file name n.h is not affected by this MAPINCLUDE because it does not end with the *from* string. In fact, the name n.h matches the *from* string exactly. Therefore, the FILE option should be omitted if your goal is to change only the n.h file name.

MAXALIGN

The MAXALIGN pragma specifies that objects of a composite type are to be given the maximum alignment supported by the compiler for the architecture of the host system. This pragma precedes the data type declaration.

```
MAXALIGN
```

There is no default setting for this pragma.

Usage Guidelines

- This pragma is only supported in TNS/E and TNS/X compilers.
- This pragma applies only to definitions of a composite type. For C, composite types include array types, struct types, and union types. For C++, composite types also include classes.
- You do not need to specify this pragma if it appears in the standard header file that supplies the data type declaration.
- The size of the type is padded at the end if necessary to make the size an integral multiple of its alignment.
- For the TNS/E compiler, the maximum alignment supported is 16 bytes.

Examples

1.
   ```
   #pragma MaxAlign
   typedef double jmp_buf[160]; // Aligned at 16 bytes
                                // instead of 8 bytes
   ```

2.
   ```
   #pragma MaxAlign
   struct s {
       int I;
       int J;
   }; // Aligned at 16 bytes, with size 16 bytes and
      // 8 bytes of padding at the end
   ```

MIGRATION_CHECK

The MIGRATION_CHECK pragma directs the TNS/R native C++ compiler to perform a migration check, to aid in migrating from VERSION2 of the Standard C++ Library to VERSION3.

```
MIGRATION_CHECK
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	Not set	Not set
G-series TNS c89 utility	Not set	Not set
TNS/R native C and C++ compilers	Not set	Not set
Native c89 utility	Not set	Not set
c99 utility	N.A.	N.A.
TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

- On Guardian environment, the `MIGRATION_CHECK` pragma must be entered on the NMCPLUS or CPPCOMP command line along with the `VERSION2` directive. On OSS environment or Windows, specify the `-Wmigration_check` option of the `c89` utility along with `-Wversion2`.
- Running a migration check does:
    - Analyses `VERSION2` source code using the compiler and header files provided beginning with G06.20
    - Issues a warning when a class or member function is used that has changed or become obsolete for `VERSION3`
- When `MIGRATION_CHECK` is specified, no object file is produced by NMCPLUS, CPPCOMP, or `c89`. The listing, if enabled, itemizes the functions called from the `VERSION2` library that are not supported in `VERSION3`.
    - On OSS environment, a listing is produced only if you also specify `-Wnosuppress` . Diagnostics are also sent to `stderr`.
    - On Guardian environment, a listing is produced by default unless you also specify `NOLIST` on the command line. The listing is directed to the hometerm unless an OUT file is specified.

- MIGRATION_CHECK must be used with the **VERSION2** on page 324 command-line directive. If it is used with **VERSION1** on page 322 or **VERSION3** on page 326 (or if no version is specified), this error is output to the home terminal and to the compilation listing if it is enabled:

  ```
  ***Error: MIGRATION_CHECK option only allowed with version2.
  ```

- The source code to be analyzed must be error-free, valid VERSION2 C++ source.
- No migration check is available for migrating from VERSION1 to VERSION3.
- Required source code changes might be either trivial or significant in migrating from VERSION2 to VERSION3.

  For example, many components available in the VERSION2 library are also available in VERSION3, sometimes by different name. Some components don't exist in VERSION3, but many of these can be easily built using VERSION3 components.

Examples

- To run a migration check in the OSS environment:

  ```
  c89 -Wversion2 -Wmigration_check -Wnosuppress myprog.cpp
  ```

- To run a migration check in the G-series Guardian environment:

  ```
  nmcplus / in myprogc, out $s.#list / ; version2, migration_check
  ```

- To run a migration check in the H-series or J-series Guardian environment:

  ```
  cppcomp / in myprogc, out $s.#list / ; version2, migration_check
  ```

MIGRATION_CHECK 32TO64

MIGRATION_CHECK 32TO64 enables additional compiler diagnostics. These warnings detect valid C/C++ code that potentially works in an unexpected fashion when code designed for ILP32 is compiled using the LP64 data model.

Usage Guidelines

- This command-line directive is supported only in TNS/E compilers running on H06.24/J06.13 RVU and later versions. On OSS and Windows environment, the compiling option is –Wmigration_check=32to64.
- MIGRATION_CHECK 32TO64 is a command-line directive that must be entered on the compiler RUN command line but not in the source text. For more information, see **LP64 Data Model** on page 506.

MULTIBYTE_CHAR

The MULTIBYTE_CHAR pragma directs the native mode C/C++ compiler to support multibyte characters in source code, specifically in comments, string literals, and character constants. When MULTIBYTE_CHAR is specified, the compiler issues a warning if an invalid multibyte character is detected. Currently SJIS is the only multibyte character set supported.

```
MULTIBYTE_CHAR
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS c89 utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set	Not set
Native c89 and c99 utilities	Not set	Not set
TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

- MULTIBYTE_CHAR is a command-line directive that must be entered on the compiler RUN command line, not in the source text.
- The MULTIBYTE_CHAR directive can also be specified with the -Wmultibyte_char flag of the c89 or the c99 utility.

NEST

The NEST pragma controls whether the compiler accepts nested comments. The NEST pragma directs the compiler to accept nested comments, recursively pairing begin-comment symbols (/*) with end-comment symbols (*/). NONEST directs the compiler not to accept nested comments.

```
[NO]NEST
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	NONEST	NONEST
G-series TNS c89 utility	NONEST	NONEST
TNS/R native C and C++ compilers	N.A.	N.A.
Native c89 and c99 utilities	N.A.	N.A.
TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	N.A.	N.A.

Usage Guidelines

- For the TNS C compiler, the NEST pragma can be entered on the compiler RUN command line or in the source text. For OSS, the pragma can be entered only in the source file.
- The native C and C++ compilers do not support the NEST pragma. The ISO/ANSI C Standard does not support nested comments.

Example

In this example, the NEST pragma enables a single pair of comment symbols to delimit three source lines that themselves contain comments:

```
#pragma NEST
/*
   r = 20;                 /* initialize r */
   r_squared = r*r;        /* square x */
   area = 3.14*r_squared; /* compute the circle's area */
*/
```

NEUTRAL

The NEUTRAL pragma is only supported for use within HPE NonStop system header files for TNS/E and TNS/X. It controls whether a struct, class, or union definition is considered as part of the C++ CPPNEUTRAL dialect and marks it as being available in the neutral C++ standard library.

```
NEUTRAL
```

This pragma is not recognized by the TNS/R compiler.

There is no default setting for this pragma.

Usage Guidelines

- The `NEUTRAL` pragma cannot appear on the RUN command line for CCOMP or CPPCOMP (H-series or J-series Guardian). This pragma can only appear in the source code.
- Only the `NEW`, `EXCEPTION`, and `TYPEINFO` headers are in the `CPPNEUTRAL` dialect. These headers define these new/delete family of storage allcocation functions:

  ```
  class type_info()
  set_unexpected()
  set_new_handler()
  set_terminate()
  class exception()
  class bad_alloc()
  class bad_cast()
  class bad_exception()
  class bad_typeid()
  ```
- Storage allocation and exception handling are common (except for the classes defined in the `stdexcept` header).

Example

See **Using the Neutral C++ Dialect** on page 102.

NLD(arg)

The `NLD` command-line directive specifies arguments to be passed to the nld utility, the linker for TNS/R native code that is not PIC (Position-Independent Code).

`NLD(`*`arg`*`)`

arg is any argument accepted by the `nld` utility. Enclose arguments in parentheses, separated by spaces.

For more details on valid syntax and semantics, see the *`nld` Manual* .

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set	Not set

Table Continued

	SYSTYPE GUARDIAN	SYSTYPE OSS
Native c89 utility	Not set	Not set
c99 utility	N.A.	N.A.
TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	N.A.	N.A.

Usage Guidelines

- On Guardian environment, the NLD pragma must be entered on the compiler RUN command line for TNS/R native C and C++. On OSS environment, specify the NLD pragma by using the -Wnld= *arg* option with the c89 utility.
- If you are linking TNS/R PIC (Position-Independent Code), you must use the **LD(arg)** on page 260 pragma to specify arguments to the ld utility.
- If you are linking TNS/E PIC (Position-Independent Code), you must use the -Weld= *arg* command-line flag to specify arguments to the eld utility.
- If you are linking TNS/X PIC (Position-Independent Code), you must use the -Wxld=*arg* command-line flag to specify arguments to the xld utility.
- The NLD pragma does not invoke the linker. To invoke nld, you must include other pragmas such as **RUNNABLE** on page 292 or **LINKFILE** on page 262. If nld is not invoked, the NLD pragma is ignored.
- You cannot use the NLD directive if you also use either of these directives:
  - **CALL_SHARED** on page 210
  - **SHARED** on page 300

Example

This command invokes the TNS/R native mode C compiler and uses the NLD directive to pass options to the nld utility:

```
nmc /in prog1/ prog1o;nld(-set highpin off -set highrequester
off),runnable
```

NOEXCEPTIONS

The NOEXCEPTIONS compiler directive directs the native C++ compiler to disable support for exceptions and exception handling.

```
NOEXCEPTIONS
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set	Not set

Native
`c89`
and
`c99`
utilities
	Not set	Not set

TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

- On Guardian environment, the `NOEXCEPTIONS` pragma can appear only on the RUN command line for NMCPLUS or CPPCOMP. On OSS environment, the `-Wnoexceptions` option can appear only on the RUN command line for the native `c89` or the `c99` utility.
- The `NOEXCEPTIONS` directive affects only native C++ programs compiled with the `VERSION2` or `VERSION3` dialects. The directive is ignored for C programs and C++ programs compiled using `VERSION1`.
- The `NOEXCEPTIONS` directive can improve application performance by removing unneeded processing steps with an application that does not use exceptions or perform exception handling.

NON_SHARED

The `NON_SHARED` pragma directs the native C and C++ compilers to generate non-shared code (that is, not PIC (Position-Independent Code).

Compare the action of `NON_SHARED` with that of the two related pragmas:

- **CALL_SHARED** on page 210 directs the compiler to generate PIC.
- **SHARED** on page 300 directs the compiler to generate PIC and to invoke the `ld` linker to create a PIC library file.

```
NON_SHARED
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	`NON_SHARED`	`NON_SHARED`
Native `c89` utility	TNS/R code: `NON_SHARED` TNS/E code: N.A. TNS/X code: N.A.	TNS/R code: `NON_SHARED` TNS/E code: N.A. TNS/X code: N.A.
`c99` utility	N.A.	N.A.
TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	N.A.	N.A.

Usage Guidelines

- On Guardian environment, the `NON_SHARED` pragma can appear only on the RUN command line for NMC or NMCPLUS. On OSS environment, the `-Wnon_shared` option can appear only on the RUN command line for the `c89` utility when the `nld` utility is used.
- On TNS/R Guardian environment, the default behavior of `NON_SHARED` is to not call the `ld` linker. The TNS/R native C or C++ driver only calls the linker if you also specify either **RUNNABLE** on page 292 or **LINKFILE** on page 262 , or if you specify **SHARED** on page 300.
  - If you specify `NON_SHARED`, the compilation results in a non-PIC linkable object file (a linkfile).
  - If you include both the `NON_SHARED` and `RUNNABLE` pragmas, the compilation results in a non-PIC executable object file (a loadfile) from the `nld` linker.
- On TNS/R OSS, the default behavior of the `-non_shared` option is to automatically call the `nld` linker.

- If you specify -Wnon_shared, the compilation results in a non-PIC executable object file (loadfile).
- If you specify the -c option with -Wnon_shared, the linker is not called, and the result is a non-PIC linkable object file (linkfile).

OLDCALLS

The OLDCALLS pragma controls how the TNS C compiler generates code for function calls.

```
[NO]OLDCALLS
```

If you specify the OLDCALLS pragma, the TNS C compiler generates code for function parameters such that the parameters are stacked with the last parameter first and the first parameter last. Versions of the TNS C compiler released prior to C00 behave this way.

If you specify NOOLDCALLS, the TNS C compiler generates code for function parameters such that the parameters are stacked with the first parameter first and the last parameter last. C00 and later versions of the TNS C compiler behave this way.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	NOOLDCALLS	NOOLDCALLS
G-series TNS c89 utility	NOOLDCALLS	NOOLDCALLS
TNS/R native C and C++ compilers	N.A.	N.A.
Native c89 and c99 utilities	N.A.	N.A.
TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	N.A.	N.A.

Usage Guidelines

- The OLDCALLS pragma can be entered on the compiler RUN command line or in the source text.
- The OLDCALLS pragma has no affect if SYSTYPE OSS is specified; the compiler ignores the pragma.

- Versions of the TNS C compiler released with or after C00 and before D20.00 created automatic prototypes for nonprototyped functions. Because this behavior violates the ISO/ANSI C standard, it is no longer supported.
- The native C and C++ compilers do not support these pragmas. Native compilers do not support B-series C language function-calling behavior.

OLIMIT

The `OLIMIT` directive specifies the maximum decimal number of basic blocks of a routine that the global optimizer is to optimize. When a routine has more basic blocks than this number, the routine is not optimized and a warning is printed.

```
OLIMIT value
```

value

is the number of basic blocks of a routine that are to be optimized. Specify a value between 0 and 32767.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set	Not set
Native `c89` utility	Not set	Not set
`c99` utility	N.A.	N.A.
TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	N.A.	N.A.

Usage Guidelines

- This directive applies to TNS/R-targeted compilations only.
- On Guardian environment, the `OLIMIT` pragma can appear only on the RUN command line for NMC or NMCPLUS . On OSS environment, the `-Wolimit=[value]` option can appear only on the RUN command line for the `c89` utility.

- When a routine has more basic blocks than *value*, the routine is not optimized and a warning message is issued.
- You must also specify one of these pragmas when using the `OLIMIT` directive:
  - On OSS environment: `-Woptimize=2` or `-O`
  - On Guardian environment: `OPTIMIZE 2`
- When `OLIMIT` is not set, an optimized routine can contain no more than 1500 basic blocks. When a routine has more basic blocks than this number, it is not optimized, but a warning message is not issued.

ONCE

The `ONCE` pragma specifies that the file containing this pragma will be compiled only once during the compilation, even if it is included multiple times.

```
ONCE
```

There is no default setting for this pragma.

Usage Guidelines

- This pragma applies to native compilations only.
- The ONCE pragma can be specified only in the source code.
- Use the `ONCE` pragma only in header files.

NOTE: Do not use the #pragma `ONCE` in a header file that contains the #pragma `SECTION`. The expected behavior of this combination is undefined.

OPTFILE

The `OPTFILE` pragma specifies an optimizer file, which contains a list of functions that are to be optimized at the level specified in the file. Functions that are not in this list are optimized at the level given in the RUN command line or in the `-Woptimize` flag of the `c89` utility or the `-On` flag of the `c99` utility, where `n` is one of 0, 1, or 2.

```
OPTFILE "file-name"
```

file-name

is the name of a text file.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS c89 utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set	Not set
Native c89 and c99 utilities	Not set	Not set
TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

- For the native C and C++ compilers, the `OPTFILE` pragma can be entered only on the compiler RUN command line. For OSS, the `-Woptfile` option pragma can be entered only in the `c89` or the `c99` command line.
- Each line of the optimizer file can contain only one function name and the optimize level (0,1, or 2) that you want for that function.
- The optimizer file can raise or lower the optimize level for the given functions; the other functions in the module are compiled at the optimization level specified in the RUN command line, if any is specified, or at the default level if no level is specified.
- The function name must be the internal name used for linking by the linker utility. Therefore, the mangled name must be used for C++ programs.

OPTIMIZE

The `OPTIMIZE` pragma controls the level to which the compiler optimizes the object code.

```
OPTIMIZE level
level:

   { 0 | 1 | 2 }
```

level

specifies the level to which the compiler optimizes the code it generates. The available optimization levels are described in Usage Guidelines.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	`OPTIMIZE 1`	`OPTIMIZE 1`
G-series TNS `c89` utility	`OPTIMIZE 1`	`OPTIMIZE 1`
TNS/R native C and C++ compilers	`OPTIMIZE 1`	`OPTIMIZE 1`
Native `c89` utility	`OPTIMIZE 1`	`OPTIMIZE 1`
`c99` utility	`OPTIMIZE 1`	`OPTIMIZE 1`
TNS/E native C and C++ compilers	`OPTIMIZE 1`	`OPTIMIZE 1`
TNS/X native C and C++ compilers	`OPTIMIZE 1`	`OPTIMIZE 1`

Usage Guidelines

- For TNS and native C and C++, the `OPTIMIZE` pragma can be only entered on the compiler RUN command line. For OSS, the pragma can be specified with the -`Woptimize` flag of the `c89` utility or the `-WO`*n* flag of the `c99` utility.
- The `OPTIMIZE` pragma affects TNS C and C++ programs:
  - Optimization level 0 disables all optimizations and yields code with relatively poor performance. It does not reduce compilation time.
  - Optimization level 1 provides intrastatement optimizations; that is, optimizations within a single statement but not across statement boundaries. Because it does not affect statement boundaries, optimization level 1 is useful when you are developing and debugging your program.
  - Optimization level 2 provides both intrastatement and interstatement optimizations. Interstatement optimizations can affect statement boundaries, which, in turn, can make debugging a program more difficult. Consequently, you should use optimization level 2 only after your program is thoroughly debugged and tested.
- The `OPTIMIZE` pragma affects native C and C++ programs:
  - Optimization level 0 disables all optimizations and therefore yields code with relatively poor performance. Optimization level 0 is useful when you are developing and debugging your program and is recommended for serious debugging. Statements are well-defined when debugging;

breakpoints and stepping occurs in a manner that the user would expect when viewing the related source.

- Optimization level 1 generates optimized code sequences. Object code compiled at optimization level 1 can be symbolically debugged; statement boundaries, however, might be blurred. The Inspect, Native Inspect, and Visual Inspect debuggers choose a sensible location when a user requests a breakpoint on a source statement, but their definition of statement boundaries does not always coincide directly with source statements. The debugger emits a warning when a process is held at a statement for which the code associated with a previous source statement has not yet executed.
- Optimization level 2 generates the most optimized code sequences. Object code compiled at optimization level 2 can be symbolically debugged with a symbolic debugger.

For information on setting TNS/R code optimization levels, see the *TNS/R Native Application Migration Guide*. For information on setting TNS/E code optimization levels, see the *H-Series Application Migration Guide*. For information on setting TNS/X code optimization levels, see the *L-Series Application Migration Guide*. See also **OPTFILE** on page 279.

OVERFLOW_TRAPS

The `OVERFLOW_TRAPS` pragma determines whether the native C and C++ compilers generate code with arithmetic overflow traps. The compiler generates code that traps (issues a signal) on arithmetic overflow if `OVERFLOW_TRAPS` is set. The compiler does not generate code that traps on arithmetic overflow if `NOOVERFLOW_TRAPS` is set.

```
[ NO ]OVERFLOW_TRAPS
```

Usage Guidelines

- The `OVERFLOW_TRAPS` pragma can be entered on the compiler RUN command line or in the source text. It can also be specified with the `-Woverflow_traps` flag of the `c89` or the `c99` utility.
- Multiple pragmas can be placed in the source code to turn on and off the generation of code that traps on arithmetic overflow.
- The `OVERFLOW_TRAPS` pragma in C mimics the overflow trapping code in pTAL and the trapping behavior of the TNS C compiler. Code that traps on integer overflow is generated only for signed arithmetic operations. Unsigned arithmetic operations always truncate. Therefore, if your code uses only unsigned arithmetic operations, the code does not trap on integer overflow.
- For more details on arithmetic overflow and signals, see the *Guardian Programmer’s Guide*

Example

```
#pragma NOOVERFLOW_TRAPS
void NoTraps (void) {
/* No trapping code generated */
}
#pragma OVERFLOW_TRAPS
void Traps (void) {
/* Generates trapping code */
}
#pragma NOOVERFLOW_TRAPS
/* Again, no trapping code generated */
```

PAGE

The PAGE pragma causes a page eject in the compiler listing and prints a page heading. The page eject occurs only when the output is being directed to a printer or spooler device.

```
PAGE [ "title-string" ]
```

title-string

specifies the title to print on each subsequent page. The title string can contain up to 61 characters.

There is no default setting for this pragma.

Usage Guidelines

- The PAGE pragma cannot appear on the command line, but it can appear at any point in the source text.
- The quotation marks enclosing *title-string* are required delimiters; they are not printed.
- If the title string is longer than 61 characters, the compiler prints only the first 61.
- PAGE takes effect only if the LIST pragma is in effect.

POOL_STRING_LITERALS

The POOL_STRING_LITERALS directive specifies that within a compilation unit multiple occurrences of the same string literal are to occupy the same storage space. The default assignments for multiple occurrences of a string literal gives them separate storage space.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS c89 utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set	Not set
Native c89 and c99 utilities	Not set	Not set

Table Continued

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

The POOL_STRING_LITERALS directive must be entered on the command line. It can also be specified with the `-Wpool_string_literals` flag of the `c89` or the `c99` utility.

POP

The `POP` pragma directs the native compilers to restore the value of certain pragmas that were stored earlier by a `PUSH` pragma.

```
POP pragma-name

pragma-name:

  { EXTERN_DATA | FIELDALIGN | LIST | OVERFLOW_TRAPS |
  REFALIGNED | WARN }
```

There is no default setting for this pragma.

Usage Guidelines

- The `POP` pragma can be entered only in the source text.
- Only the values EXTERN_DATA, `FIELDALIGN` , LIST, OVERFLOW_TRAPS, REFALIGNED, and WARN can be used as arguments to the `POP` pragma. The use of any other pragma as an operand is flagged as an error.
- Each `POP` pragma has a separate stack that holds up to 32 values.

PROFDIR

The PROFDIR option specifies where an instrumented process will create the raw data file.

```
PROFDIR "pathname"
```

The default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	N.A.	N.A.

Native
`c89`
and
`c99`
utilities
	N.A.	Current working directory

TNS/E native C and C++ compilers	Default subvolume	N.A.
TNS/X native C and C++ compilers	Default subvolume	N.A.

When PROFDIR is specified, either **PROFGEN** on page 287 or **CODECOV** on page 214 must also be specified; otherwise, PROFDIR is ignored. The specified path name is used exactly as specified. If the application will run in the Guardian environment, specify a Guardian subvolume. If the application will run in the OSS environment, specify an OSS directory.

Usage Guidelines

- The PROFDIR option can be entered only on the compiler RUN command. It can also be specified with the `-Wprofdir` flag of the `c89` or the `c99` command in the OSS and Windows environments.
- The PROFDIR option is intended to be used for code profiling; it is ignored unless the **PROFGEN** on page 287 or **CODECOV** on page 214 option is also specified.
- If a program consists of more than one compilation module, you must either use the default default pathname for each module or specify the same pathname for each module.
- When code that was generated from separate compilations will later be included in a single application, you should specify the same raw data file location for all the compilations.
- In the OSS and Windows environments, the -`Wprofdir` flag has an effect only for `–Wtarget=ipf` or `–Wtarget=tns/e` or `–Wtarget=tns/x` . It is ignored (and no diagnostic is issued) for – `Wtarget=mips` or -`Wtarget=tns/r`.

For more information on code profiling and the Code Profiling Utilities, see the *Code Profiling Utilities Manual*.

PROFILING/NOPROFLING

The profiling pragma directs the TNS/X native compiler to perform profiling related code instrumentation or optimization on individual functions.

```
#pragma profiling | noprofiling
```

With the two pragmas, the original source code is divided into multiple logical scopes with profiling on or off in each scope.

The directive “#pragma profiling” ends the previous scope and starts a new scope with profiling on.

The directive “#pragma noprofiling” ends the previous scope and starts a new scope with profiling off. Compiler only looks at the state of the profiling scope where the function is defined to set the profiling flag for the function.

Scope	PROFGEN/CODECOV	PROFUSE
Profiling on	The function is instrumented so that dynamic profiling information is collected for code coverage or PGO.	The function is optimized with the dynamic profiling information and the PGO bit in procinfo is set.
Profiling off	The function is not instrumented.	The function is not optimized with dynamic profiling information. The PGO bit in procinfo is cleared.

The default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	N.A.	N.A.

Native
`c89`
and
`c99`
utilities
	N.A.	N.A.

TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	Not set	Not set

PROFGEN

The PROFGEN option directs the compiler to generate instrumented object code for use in performing profile-guided optimization. The PROFGEN option is supported by the TNS/E or TNS/X native compilers.

```
PROFGEN
```

The profile generation on individual functions are also controlled by “#pragma profiling” and “#pragma noprofiling”. For more information, see **PROFILING/NOPROFLING** on page 286.

The default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	N.A.	N.A.

Native
`c89`
and
`c99`
utilities
	Not set	Not set

TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

- The PROFGEN option can be entered only on the compiler RUN command. It can also be specified with the `-Wprofgen` flag of the `c89` or the `c99` command in the OSS and Windows environments.
- If you specify the PROFGEN option, you must also compile at optimization level 2 (OPTIMIZE 2); otherwise, PROFGEN has no effect.
- If you specify both PROFGEN and CODECOV, CODECOV overrides PROFGEN.
- In the OSS and Windows environments, the `-Wprofgen` flag has an effect only for `–Wtarget=ipf` or `–Wtarget=tns/e` or `–Wtarget=tns/x`. It is ignored (and no diagnostic is issued) for `–Wtarget=mips` or `-Wtarget=tns/r`.
- Instrumented object code can significantly increase both the compile time and execution time required by a program. Therefore, the PROFGEN option should be used only in a test environment.

For more information on code profiling and the Code Profiling Utilities, see the *Code Profiling Utilities Manual*.

PROFUSE

The PROFUSE option directs the compiler to generate optimized object code based on information in a Dynamic Profiling Information (DPI) file. The PROFUSE option is supported only by the TNS/E native compilers.

```
PROFUSE ["pathname"]
```

The profiling optimization on individual functions are also controlled by “#pragma profiling” and “#pragma noprofiling”. For more information, see **PROFILING/NOPROFLING** on page 286.

The default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	N.A.	N.A.

Native
`c89`
and
`c99`
utilities
	N.A.	File name
		`pgopti.dpi`
		in the current working directory or file name
		`pgodpi`
		in the default subvolume

TNS/E native C and C++ compilers	File name `pgodpi` in the default subvolume	N.A.
TNS/X native C and C++ compilers	File name `pgodpi` in the default subvolume	N.A.

Usage Guidelines

- The PROFUSE option can be entered only on the compiler RUN command. It can also be specified with the `-Wprofuse` flag of the `c89` or the `c99` command in the OSS and Windows environments.
- If you specify the PROFUSE option, you must also specify the OPTIMIZE 2 option; otherwise, the PROFUSE option is ignored.

- You cannot specify the PROFUSE and **PROFGEN** on page 287 options on the same command line. To do so results in an error. These two activities (generating the instrumented object file and compiling for profile-guided optimization) must occur in separate compilations.
- The PROFUSE option can optionally specify the file path of a DPI file. If a file path is not specified, the compiler looks for the file in the current working directory or default subvolume, and the file name defaults to:
  - `pgodpi`

    if the compilation is done in the Guardian environment, or in the OSS environment and the current working directory is a Guardian subvolume.
  - `pgopti.dpi`

    if the compilation is done in the Windows environment, or in the OSS environment and the current working directory is an OSS directory.

For more information on code profiling and the Code Profiling Utilities, see the *Code Profiling Utilities Manual*.

PUSH

The `PUSH` pragma directs the compiler to save the value of certain pragmas. The values can be restored later by a `POP` pragma.

```
PUSH pragma-name

pragma-name:

  { EXTERN_DATA | FIELDALIGN | LIST | OVERFLOW_TRAPS |
  REFALIGNED | WARN }
```

There is no default setting for this pragma.

Usage Guidelines

- The `PUSH` pragma can be entered only in the source text.
- Only the values EXTERN_DATA, `FIELDALIGN`, LIST, OVERFLOW_TRAPS, REFALIGNED, and WARN can be used as operands to the `PUSH` pragma. The use of any other pragma as an operand is flagged as an error.
- Each `PUSH` pragma has a separate stack, each of which holds up to 32 values.

REFALIGNED

The `REFALIGNED` pragma specifies the default reference alignment for pointers in native C and C++ programs.

```
REFALIGNED value [ pointer-name-list ]
value:
        { 2 | 8 }
pointer-name-list:
        pointer-name [ pointer-name-list ]
pointer-name:
        { type-name | variable-name }
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS c89 utility	N.A.	N.A.
TNS/R native C and C++ compilers	REFALIGNED 8	REFALIGNED 8
Native c89 and c99 utilities	REFALIGNED 8	REFALIGNED 8
TNS/E native C and C++ compilers	REFALIGNED 8	REFALIGNED 8
TNS/X native C and C++ compilers	REFALIGNED 8	REFALIGNED 8

Usage Guidelines

- The REFALIGNED pragma can be entered on the compiler RUN command line or in the source text, or it can be specified with the -Wrefalign flag of the c89 or the c99 utility.
- In native mode, if you specify REFALIGNED on the command line, you cannot include a *pointer-name-list*. A list is permitted only when the pragma appears in the source text.
- The default code generation for pointer dereferencing operations (REFALIGNED 8) expects the pointer to contain an address that satisfies the alignment requirements of the object being pointed to.

  For example, a 4-byte object should have an address that is a multiple of 4. If the object is at an address that does not satisfy its alignment requirements, the default code generation causes a compatibility trap. To avoid this, specify REFALIGNED 2 on the pointer to the object. This results in code generation that assumes the dereferenced object is not properly aligned and compensates for the misalignment.
- Pragma REFALIGNED without a *pointer-name-list* specifies the default alignment for the entire compilation and can be used in the command line or in the source text.
- A global REFALIGNED pragma affects only pointers that do not point to classes, structs, or unions.

- If you declare a TYPEDEF and it is of a type that has been given an explicit reference alignment, that REFALIGNED value is propagated to the new TYPEDEF. Otherwise, the new TYPEDEF gets the current global default REFALIGNED value.
- When SHARED2 substructs are nested within AUTO, PLATFORM, or SHARED8 structs, the substruct contributes only a 1-byte or 2-byte alignment requirement to the overall struct, even when the substruct contains 4-byte or 8-byte field types.

REMARKS

The REMARKS pragma directs the native C and C++ compilers to issue remarks.

```
REMARKS
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS c89 utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set	Not set
Native c89 and c99 utilities	Not set	Not set
TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

- The REMARKS pragma can be entered only on the compiler RUN command line. For OSS, the pragma can be specified with the -Wremarks flag of the c89 or the c99 utility.
- Remarks are informative diagnostics that are less severe than warnings.

RUNNABLE

The `RUNNABLE` pragma directs the compiler to generate an executable object file instead of a linkable object file for a single-module program.

```
RUNNABLE
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	Not set	Not set
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set	Not set

Native
`c89`
and
`c99`
utilities
	N.A.	N.A.

TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

- The `RUNNABLE` pragma can be specified only on the compiler RUN command line for the native C and C++ compilers.
- If you specify `RUNNABLE` in a translation unit that does not define the function `main`, the compiler generates a warning and produces a linkable, not executable, object file.
- The `RUNNABLE` pragma has no effect if the compiler is run from the OSS environment. The compiler ignores the pragma.
- When you specify the `RUNNABLE` pragma, the native C and C++ compilers does:
    - Invoke the appropriate linker for conventional or PIC (Position-Independent Code)
    - Specify the $SYSTEM.SYSTEM.LIBCOBEY command file to a TNS/R linker

- – The LIBCOBEY file directs the linker to link to a set of standard shared run-time libraries (SRLs). For most C and C++ TNS/R programs, this set of libraries is sufficient to create an executable program.
- – If your program requires libraries not specified in LIBCOBEY (such as the Tools.h++ library) you can direct the linker to search additional libraries using the `LINKFILE` pragma. For more details, see **LINKFILE** on page 262.

- For information on linking TNS/R C and C++ programs, see **Linking a TNS/R Module** on page 376.
- For information on linking TNS/E C and C++ programs, see **Linking a TNS/E Module** on page 392.
- For information on linking TNS/X C and C++ programs, see **Linking a TNS/X Module** on page 413.

RUNNAMED

The `RUNNAMED` pragma specifies that the object file runs as a named process, even if you do not specify the NAME option in the RUN command.

```
RUNNAMED
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	Not set	Not set
G-series TNS `c89` utility	Not set	Not set
TNS/R native C and C++ compilers	Not set	Not set

Native
`c89`
and
`c99`
utilities
	Not set	Not set

TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

- The `RUNNAMED` pragma can appear only on the compiler RUN command line for native C and C++. For TNS compilers, the pragma can only be specified with the `-Wrunnamed` flag of the `c89` or the `c99` utility.
- The `RUNNAMED` attribute is set for native C and C++ programs only if an executable object file is the output of the compilation. (Process attributes cannot be set for native relinkable object files.)
- For TNS programs, you can set the RUNNAMED object-file attribute either during compilation using the `RUNNAMED` pragma or after compilation using the Binder SET command.

  Using the `RUNNAMED` pragma:

  - Enter the `RUNNAMED` pragma anywhere in the source text or in the RUN command that executes the compiler.
  - When you bind several object files together, Binder sets the RUNNAMED attribute in the target object file if any one of the object files has its RUNNAMED attribute set.

  Using the Binder SET command:

  - Specify the Binder SET RUNNAMED ON command. The default state for the Binder is RUNNAMED OFF.
  - The Binder SET RUNNAMED ON command takes precedence over any RUNNAMED pragmas that might or might not have been specified in individually compiled object files.

- For native programs, you can set the HIGHREQUESTERS object-file attribute either during compilation using the `HIGHREQUESTERS` pragma or after compilation using a linker utility.
- A named process running at a high PIN can be opened with the Guardian OPEN system procedure by an unconverted process running at a low PIN. For example, a named server process can be opened by an unconverted requester process.

  An unconverted process is one that has not been modified to use any of the extended features of the D-series Guardian procedures. Unconverted processes can run only at a low PIN, while converted processes can run at either a low PIN or a high PIN. For more details, see the *Guardian Programmer's Guide.*

- When a named process is deleted, it prompts the operating system to send a message to the process that created it. For example, a process that creates server processes needs to be informed if one of the created processes is deleted. Similarly, a process that manages a batch job needs to be informed about the survival of processes within the job.

  The operating system sends a message when a named process is deleted as a result of either normal process termination or a processor failure. The operating system sends a message when an unnamed process is deleted only as a result of a normal process termination. It is up to the process that created the unnamed process to check for processor failure. For more details on creating and managing processes, see the *Guardian Programmer's Guide*.

RVU

The RVU pragma sets the value of the __G_SERIES_RVU or __H_SERIES_RVU or __L_SERIES_RVU feature-test macro. These feature-test macros are used in HPE NonStop standard header files to control whether declarations that depend on a specific RVU are available.

```
RVU { g-series-rvu | h-series-rvu | l-series-rvu }
```

g-series-rvu

sets the __G_SERIES_RVU feature-test macro to the specified value. The value has the form G06.nn. When specified for a C module compilation, this option also causes the compiler to issue an error, instead of a warning, for implicitly declared functions.

h-series-rvu

sets the __H_SERIES_RVU feature-test macro to the specified value. The value has the form H06.nn. When specified for a C module compilation, this option also causes the compiler to issue an error, instead of a warning, for implicitly declared functions.

l-series-rvu

sets the __L_SERIES_RVU feature-test macro to the specified value. The value has the form Lyy.mm, where yy is the year and mm is the month of the applicable RVU. When specified for a C module compilation, this option also causes the compiler to issue an error, instead of a warning, for implicitly declared functions.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS c89 utility	N.A.	N.A.
TNS/R native C and C++ compilers	The RVU in which the compiler was last released	The RVU in which the compiler was last released
G-series native c89 utility	The RVU in which the compiler was last released	The RVU in which the compiler was last released
H-series or J-series native c89 or c99 utility	The H-series RVU in which the compiler was last released	The H-series RVU in which the compiler was last released

Table Continued

	SYSTYPE GUARDIAN	SYSTYPE OSS

L-series native
`c89`
or
`c99`
utility
	The L-series RVU in which the compiler was last released	The L-series RVU in which the compiler was last released

TNS/E native C and C++ compilers	The RVU in which the compiler was last released	The RVU in which the compiler was last released
TNS/X native C and C++ compilers	The RVU in which the compiler was last released	The RVU in which the compiler was last released

Usage Guidelines

- For the TNS/E native compilers in the Guardian environment, only the *h-series-rvu* option is supported.
- For the TNS/X native compilers in the Guardian environment, only the *l-series-rvu* option is supported.
- For the TNS/R native compilers in the Guardian environment, only the *g-series-rvu* option is supported.
- On the G-series version of the `c89` command (PC and OSS environment), only the *g-series-rvu* option is supported.
- On the H-series version of the `c89` command, you can specify either *g-series-rvu* or *h-series-rvu* , but the option you specify must be compatible with the option specified for the TARGET pragma. That is, you can specify *g-series-rvu* only with `-Wtarget=tns/r`, *h-series-rvu* only with `-Wtarget=tns/e`, and *l-series-rvu* only with `-Wtarget=tns/x`.
- No checking is performed to determine whether the specified RVU actually exists.
- The RVU pragma can appear only on the compiler RUN command line for native C and C++.

SAVEABEND

The `SAVEABEND` pragma controls whether the system creates a save file if the program terminates abnormally during execution. The `SAVEABEND` pragma specifies that the system is to create a save file if the program terminates abnormally. `NOSAVEABEND` specifies that the system is not to create a save file.

```
[NO]SAVEABEND
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	NOSAVEABEND	NOSAVEABEND
G-series TNS c89 utility	NOSAVEABEND	NOSAVEABEND
TNS/R native C and C++ compilers	NOSAVEABEND	NOSAVEABEND
Native c89 and c99 utilities	NOSAVEABEND	NOSAVEABEND
TNS/E native C and C++ compilers	NOSAVEABEND	NOSAVEABEND
TNS/X native C and C++ compilers	NOSAVEABEND	NOSAVEABEND

Usage Guidelines

- For native C and C++ you can specify the SAVEABEND pragma only on the compiler RUN command line. You can also specify the -W[no]saveabend flag of the c89 or the c99 utility.
- On TNS, the pragma can only be specified with the -W[no]saveabend flag.
- The SAVEABEND attribute is set for native C and C++ programs only if an executable object file is the output of the compilation. (Process attributes cannot be set for native relinkable object files.)
- For TNS programs, the last SAVEABEND or NOSAVEABEND pragma in a translation unit determines whether or not the system is to create a save file.
- You can set the SAVEABEND object-file attribute after compilation using the Binder SET command (for TNS programs) or a linker utility (for native programs).
- For this option to be effective at run time, the symbolic debugger must be available on the system on which the process runs.
- The INSPECT and SAVEABEND pragmas are related:
  - If you specify NOINSPECT, the compiler automatically disables SAVEABEND (as though you had explicitly specified NOSAVEABEND).
  - If you specify SAVEABEND, the compiler automatically enables the symbolic debuggers (as though you had explicitly specified INSPECT).
- The save file contains data area and file-status information at the time of failure. You can examine the save file during a symbolic debugger session. The symbolic debugger assigns the save file a name of the form ZZSA *nnnn*, where *nnnn* is an integer. The default names for volume and subvolume are the

object file's volume and subvolume. (You can specify a name for the save file using the symbolic debugger.) For information on the save file, see the:

- *Inspect Manual*
- *Native Inspect Manual*

SEARCH

For TNS programs, the `SEARCH` pragma directs the Binder to search a given object file when attempting to resolve external references in a program compiled with the `RUNNABLE` pragma.

For native programs compiled with the `RUNNABLE` pragma, the `SEARCH` pragma directs the `nld` utility (for conventional code) or the `eld`, `xld`, or `ld` utility (for PIC code) to link in the entire given object file (not just the procedure that contains the external reference). For more details, see the:

- *eld and xld Manual*
- *ld Manual*
- *nld Manual*

```
SEARCH "object-file"
```

object-file

specifies the name of the object file to be searched.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	Not set	Not set.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set	Not set

Native
`c89`
and
`c99`
utilities
	N.A.	N.A.

TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

- On Guardian environment, the `SEARCH` pragma must be entered on the compiler RUN command line.
- The quotation marks enclosing *object-file* are required delimiters.
- If the `SEARCH` pragma is specified in the same compilation unit with the `XMEM`, `NOXMEM`, `WIDE`, or `SYMBOLS`pragmas, the `SEARCH` pragma must be specified after these pragmas.

SECTION

The `SECTION` pragma gives a name to a section of a source file for use in an `#include` directive. Sections enable you to include only a portion of a file.

```
SECTION sec-name
```

sec-name

is a valid C identifier to associate with all source text that follows the `SECTION` pragma until another `SECTION` pragma or the end of the source file.

There is no default setting for this pragma.

Usage Guidelines

- The `SECTION` pragma can appear only in the source text.
- A section terminates at the next `SECTION` pragma or, if there are no more `SECTION` pragmas, at the end of the file.

NOTE: Do not use the #pragma `ONCE` in a header file that contains the #pragma `SECTION`. The expected behavior of this combination is undefined.

Example

This example shows that `file1` contains three sections:

```
#pragma SECTION namea
/* ... */
#pragma SECTION nameb
/* ... */
#pragma SECTION namec
```

To use these sections so that `fileA` contains sections `namea` and `namec`, and `fileB` contains section `nameb`, `fileA should contain`:

```
#include "file1(namea, namec)”
```

and `fileB should contain`:

```
#include "file1(nameb)"
```

SHARED

The SHARED pragma directs the native C and C++ compilers to generate shared code, which is PIC (Position-Independent Code) and to invoke the ld or eld linker to create a PIC library file, or a dynamic-link library (DLL).

Compare the action of SHARED with that of the two related pragmas:

- **CALL_SHARED** on page 210 directs the compiler to generate PIC.
- **NON_SHARED** on page 275 directs the compiler to generate a non-PIC loadfile that cannot be shared and cannot access PIC files.

```
SHARED
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS c89 utility	N.A.	N.A.
TNS/R native C and C++ compilers	NON_SHARED	NON_SHARED
Native c89 and c99 utilities	TNS/R code: NON_SHARED TNS/E code: CALL_SHARED	TNS/R code: NON_SHARED TNS/E code: CALL_SHARED
TNS/E native C and C++ compilers	CALL_SHARED	CALL_SHARED
TNS/X native C and C++ compilers	CALL_SHARED	CALL_SHARED

Usage Guidelines

- On Guardian environment, the SHARED pragma can appear only on the RUN command line for CCOMP, CPPCOMP, NMC, or NMCPLUS. On OSS environment, the -Wshared option can appear only on the command line for the c89 or the c99 utility.
- The default behavior of SHARED is to invoke the eld or ld linker and to produce a PIC library file (a DLL).

  - On Guardian environment, if you specify both SHARED and RUNNABLE , the compiler issues an error and terminates.
  - On OSS environment, if you specify both -c and -Wshared, the compiler issues an error and terminates.

- If you specify CPPONLY or SYNTAX with the SHARED pragma, linking does not occur because it is prevented by the additional pragmas.
- For complete information about programming with DLLs and linker options for DLLS, see the:
  - *DLL Programmer's Guide for TNS/R Systems*
  - *DLL Programmer's Guide for TNS/E and TNS/X Systems*
  - *eld and xld Manual*
  - *ld Manual*
  - *rld Manual*
- The SHARED pragma cannot be used with the **CALL_SHARED** on page 210, **SRL** on page 305 or **NON_SHARED** on page 275 pragmas. A warning is issued if these pragmas are combined.
- EXTERN_DATA gp_ok is not compatible with generation of shared code. The compiler issues a warning if this pragma is combined with the **SHARED** on page 300 pragma or **CALL_SHARED** on page 210, and ignores the gp_ok directive.

SQL

The SQL pragma enables the compiler to process subsequent SQL statements. Embedded SQL/MP statements can be placed in C code but not in C++ code.

```
SQL [ sql-option | ( sql-option [ , sql-option ]... ) ]

sql-option:
   { [NO]WHENEVERLIST              )
   { [NO]SQLMAP                    }
   { RELEASE1 | RELEASE2           }
   { CHAR_AS_ARRAY | CHAR_AS_STRING }
   { CPPSOURCE "filename"          }
```

[NO]WHENEVERLIST

controls whether the compiler listing includes notification of active SQL WHENEVER clauses.

WHENEVERLIST directs the compiler to provide a summary list of the WHENEVER clauses that are active when an SQL statement is processed. NOWHENEVERLIST directs the compiler to omit this summary list.

If you do not specify either WHENEVERLIST or NOWHENEVERLIST, the C compiler assumes NOWHENEVERLIST.

[NO]SQLMAP

controls whether the compiler listing includes a map that associates SLT indexes with embedded SQL statements. This map is useful when you use the Measure product to measure the execution time of embedded SQL statements. For more details regarding SLT indexes, see the *Measure Reference Manual*.

`SQLMAP` directs the compiler to include the map of SLT indexes, and `NOSQLMAP` directs the compiler to omit this map.

If you do not specify either `SQLMAP` or `NOSQLMAP`, the C compiler assumes `NOSQLMAP`.

RELEASE1 | RELEASE2

specifies the earliest release of NonStop SQL/MP that supports the features used in the embedded SQL statements. If you do not specify either `RELEASE1` or `RELEASE2`, the C compiler uses the version of SQL that is installed on the compilation system. The latest current version of SQL is 345.

RELEASE1

specifies that the embedded SQL statements require features first available in NonStop SQL Release 1. Consequently, the resultant program can execute on a system that supports either Release 1 or Release 2 of NonStop SQL. This option cannot be used for native C programs.

RELEASE2

specifies that the embedded SQL statements require features first available in NonStop SQL Release 2. Consequently, the resultant program can execute only on a system that supports Release 2 of NonStop SQL.

CHAR_AS_ARRAY | CHAR_AS_STRING

specifies whether a C string that is used as an SQL host variable contains an extra byte for the null terminator. `CHAR_AS_STRING` is the default setting.

CHAR_AS_ARRAY

specifies a C string without the extra byte.

CHAR_AS_STRING

specifies a C string with the extra byte for the null terminator.

CPPSOURCE "*filename*"

directs the TNS C compiler to generate a preprocessed C source file with the given file name. *filename* must be a valid Guardian file name. The generated file is empty except for SQL source RTDUs. When using this flag, a macro expansion cannot contain any part of an SQL statement and the `RELEASE1 SQL` pragma option cannot be used. This option is intended primarily for use with the Distributed Workbench Facility (DWF). Note that `CPPSOURCE` is not supported for native mode C.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	Not set	Not set
G-series TNS `c89` utility	Not set for C, N.A. for C++	Not set for C, N.A. for C++
TNS/R native C and C++ compilers	Not set for C, N.A. for C++	Not set for C, N.A. for C++

Table Continued

	SYSTYPE GUARDIAN	SYSTYPE OSS
Native c89 utility	Not set for C, N.A. for C++	Not set for C, N.A. for C++
c99 utility	N.A.	N.A.
TNS/E native C and C++ compilers	Not set for C, N.A. for C++	Not set for C, N.A. for C++
TNS/X native C and C++ compilers	Not set for C, N.A. for C++	Not set for C, N.A. for C++

Usage Guidelines

- For the native compilers, the SQL pragma can appear on the Guardian compiler RUN command line or be specified with the -Wsql or -Wsqlcomp flag of the c89 utility. For the TNS C compiler, the SQL pragma can be specified at the start of the source text, before any declarations or definitions except comments.
- With the SQL pragma,exec and sql become C keywords. Note that decimal is a predefined data type.
- For native C, the SQL pragma implicitly specifies the EXTENSIONS pragma and defines _TANDEM_SOURCE.
- For native mode C, RELEASE1 cannot be specified.
- For native mode C, the CPPSOURCE option is not supported.
- If you specify the IEEE_FLOAT pragma, you cannot also specify the SQL pragma.
- For TNS/E and TNS/X targeted compilations, specifying the SQL pragma overrides the floating-point default and TANDEM_FLOAT is used.
- For compiling embedded SQL using the c89 utility in the OSS environment, there are two related flags:
  - Use -Wsql to compile a source file containing embedded SQL.
  - Use -Wsqlcomp -o to run the SQLCOMP compiler on an executable file containing SQL (specify the executable file with the -ooption).

  You can compile, link, and run SQLCOMP on a program in one step by specifying both of these flags in one command line.
- For more details about the interface from a C program to a NonStop SQL/MP database, see the *SQL/MP Programming Manual for C.*

- The `SQL` pragma pertains only to SQL/MP, and not to SQL/MX. For more details about the interface from a native C or C++ program to a NonStop SQL/MX database, see the *SQL/MX Programming Manual for C and COBOL.*
- For PC compilations, many more options exist. See the documentation for the apppropriate PC-based compilation tool.

SQLMEM

The `SQLMEM` pragma provides the ability to alter the placement of SQL data structures from extended memory to the user data segment.

The default setting is `SQLMEM EXT`.

```
SQLMEM { USER | EXT }
```

USER

directs the compiler to place the SQL data structures in the user data segment, which is the global area addressable in 16 bits. The `SQLMEM USER` pragma improves the access time to the data structures but should be used judiciously, because it can exhaust the user data segment.

EXT

directs the compiler to place the SQL data structures in the extended segment.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	Not set	Not set
G-series TNS `c89` utility	Not set for C, N.A. for C++	Not set for C, N.A. for C++
TNS/R native C and C++ compilers	N.A.	N.A.

Native
`c89`
and
`c99`
utilities
	N.A.	N.A.

TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	N.A.	N.A.

Usage Guidelines

- The SQLMEM pragma can be entered on the compiler RUN command line or in the source text. You can use the SQLMEM pragma as many times as necessary in the program.
- You can specify the SQLMEM pragma only if the XMEM and SQL pragmas have been previously specified.
- The native C and C++ compilers do not support this pragma. Native process memory architecture does not require its use.

SRL

The SRL pragma specifies that a module is being compiled to link into a TNS/R native user library. (A TNS/R native user library is a private shared run-time library (SRL).

```
SRL
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS c89 utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set	Not set
Native c89 utility	Not set	Not set
c99 utility	N.A.	N.A.
TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	N.A.	N.A.

Usage Guidelines

- For the TNS/R native compilers, the SRL pragma can be entered only on the compiler RUN command line. In the OSS environment, the pragma can be specified with the -Wsrl flag of the c89 utility.
- This pragma signifies that the code is being compiled to go into a SRL. This prevents the compiler from using GP-relative addressing for global data. (Global data can be either extern or static in scope.) This also causes the compiler to put anonymous literals into the read-only data area.

- Customers cannot use this pragma to build a public shared run-time library (SRL).
- This pragma is not valid for the TNS/E and TNS/X native compilers.

SRLExportClassMembers

The `SRLExportClassMembers` pragma controls which members of a class are exported from an SRL or a user library (a private shared run-time library). This pragma is supported for TNS/R native C and C++ only.

```
SRLExportClassMembers srl_id exports
```

srl_id

is the name of the SRL that exports the class.

exports

PROTECTED | PUBLIC | * | (item_list)

item_list:

item | item_list , item

item:

is a member of the class.

This pragma has no default settings.

Usage Guidelines

- The `SRLExportClassMembers` pragma is used to specify which members of a class are exported from an SRL. Exporting a member function from an SRL means that the member function definition is provided by the SRL. The SRL's clients should never have a copy (inlined or non-inlined) of the function definition within the client's code. A client's call of the member function will always resolve to calling the member function within the SRL.

  The "same" `SRLExportClassMembers` pragma must appear in the class definition for both the compilation of the SRL and the compilation of the SRL client. The results of inconsistencies between the SRL and the SRL clients are undefined. When the `SRLExportClassMembers` pragma is processed while compiling a client of the SRL, the compiler never inlines a call to a member function exported by the SRL.

- If *srl_id* matches the name given by `#pragma SRLName`, this pragma causes the compiler to generate code for the exported definition, otherwise the compiler treats the definition as a declaration.
- The `SRLExportClassMembers` pragma must appear within the class definition. The class definition must be in a global (file) scope.
- Non-static data members cannot be exported.
- The `SRLExportClassMembers` pragma applies only to methods introduced by the class, not to methods inherited by the class.
- The `SRLExportClassMembers PROTECTED` exports all protected members of the class whose declaration textually precedes the pragma.
- `SRLExportClassMembers PUBLIC` exports all public members of the class whose declaration textually precedes the pragma.

- SRLExportClassMembers * exports all protected and all public members of the class whose declaration textually precedes the pragma.
- This pragma is not valid for the TNS/E and TNS/X native compilers.

SRLExports

The SRLExports pragma specifies that a definition of an external function or external variable is exported from an SRL or a user library (a private shared run-time library). This pragma sets the srl_export bit for the definition in the object file. When this object file is linked, nld automatically adds any items which have this bit set to the export list of the SRL. The SRLExports pragma is supported for TNS/R native C and C++ only.

```
SRLExports
```

There is no default setting for this pragma.

Usage Guidelines

- This pragma is valid only for TNS/R-targeted compilations.
- The SRLExports pragma is only allowed in the source file. The SRLExports pragma must appear directly before the declaration of the external function or external variable. To have an effect, the corresponding definition must also be within the same compilation unit.
- Normally constants which have compile-time computable values are allocated in the read-only data section. However, if the constant is exported by the SRLExports pragma, the constant is allocated in the large data section. This is because read-only data is not allowed to be exported by an SRL.
- Pragma SRLExports cannot apply to static objects or static functions, or class members.
- Pragma SRLExportClassMembers can be used to export class members.

Example

```
#pragma srlexports
extern int foo (void) {
   ...
}

#pragma srlexports  // Warning, Not applicable for static
                    // functions.
static int StaticFunction (void);
```

SRLName

The SRLName pragma is used to "name" an SRL or a user library (a private shared run-time library). This pragma is supported for TNS/R native C++ only.

```
SRLName srl_id
```

srl_id

is the name of the SRL.

There is no default setting for this pragma.

Usage Guidelines

- This pragma is valid only for TNS/R-targeted compilations.
- The SRLName pragma is used to determine whether a SRLExportClassMembers pragma provides a definition or a declaration for its exported functions.
- The SRLName pragma must appear in the source file before any SRLExportClassMembers pragma.

Example

```
// inc.h
struct s {
  s(){};
  foo(){};
  static int a,b,c; int d;
  goo(){};
  goo(int){};
  static static_fcn(){};

#pragma SRLExportClassMembers  SRL_1 \
        (s(), foo(), goo(), goo(int))
};

// SRL_1.C
// This provides the code for the exported member
// functions of s.

#pragma SRLName SRL_1

#include "inc.h"  // This causes the member function of
                  // s to be defined
...
```

To compile:

```
 c89 -Wsrl -c SRL_1.C
// client.C
#include "inc.h"
// Member functions exported by s are
// only declared - no definition is provided.
```

To compile:

```
c89 client.C SRL_1.o
```

SSV

The SSV pragma specifies a list of search subvolumes (SSVs) to be searched for files specified in #include directives.

In general, the subvolumes are searched in the order specified in the search subvolume list, starting with SSV0, then SSV1, and so on.

```
SSVn { "[node.]$volume"          }
     { "[$volume.]subvolume"     }
```

```
        { "[node.]$volume.subvolume"}
```

n

is the order in which to search the subvolumes; *n* is a number in the range 0 through 49.

node

is the name of the node to search.

volume

is the name of the volume to search.

subvolume

is the name of the subvolume to search.

The pragma default settings `are`:

	SYSTYPE GUARDIAN	**SYSTYPE OSS**
TNS C compiler	Current subvolume and then compiler subvolume	Current subvolume and then compiler subvolume
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	Current subvolume and then compiler subvolume	Current subvolume and then compiler subvolume

Native
`c89`
and
`c99`
utilities
	N.A.	N.A.

TNS/E native C and C++ compilers	Current subvolume and then compiler subvolume	Current subvolume and then compiler subvolume
TNS/X native C and C++ compilers	Current subvolume and then compiler subvolume	Current subvolume and then compiler subvolume

Usage Guidelines

- The `SSV` pragma can be entered on the compiler RUN command line or in the source text.
- The `SSV` pragma cannot be used if the compiler is run from the OSS environment. The compiler issues a warning.
- On the PC platform, you can perform the function of the SSV pragma (specifying the directories to be searched) on the Project Directories page, available from the Options menu. Enter the search path in the Include box.

- The enclosing quotation marks are required delimiters around the node, volume, and subvolume specifications.
- No leading zeros are allowed in the `SSV` number. Therefore, a specification such as SSV09 is invalid.
- The maximum number of search subvolumes that can be specified in a compilation is 50. However, the maximum number of characters in the TACL command buffer is 239 characters. Therefore, it is unlikely that 50 SSVs can be specified on a TACL command line because of the buffer size limitation.
- `SSVs` must be specified in ascending sequential order; that is, `SSV0`, `SSV1`, `SSV2`, `SSV3`, and so on. If a number is skipped, the TNS C compiler and Cfront ignore the remaining SSVs; the native compilers, however, process the remaining `SSVs`.
- Using the native compilers, if you specify `SSV` pragmas in a source file instead of on the command line, the numeric order (*n*) is ignored entirely and the`SSV` pragmas are processed in the order encountered.

  For example, in a source file you can specify SSVs for the native compilers without regard to numeric order:

  ```
  SSV0 "$myvol.subvol0", SSV3 "$myvol.subvol3", SSV4 "$myvol.subvol4"
  ```

  However, if you enter the preceding `SSV` pragma list on the command line, only the first `SSV` is set; the others are ignored because of the numeric gap.

- The native compilers also support the setting of SSVs using ASSIGN statements. If you use both ASSIGNs and command-line `SSV` pragmas, the two sets are unioned together. If the two sets have any duplicate SSV numbers, the SSV specified in the command line is used.
- If you use `SSV` pragmas with TNS C, the compiler subvolume is not searched. Consequently, if your program needs any standard header file, you must add a search subvolume (SSV) for the system library.

  For example, if you specify any `SSV` pragmas and your program contains a standard library file header such as `#include <stdioh>`, you must have an `SSV` pragma for “`$system.system`” (the location of the`stdioh` file on the NonStop host), because the compiler does not automatically search the compiler subvolume.

- If you specify `SSV` pragmas, the compiler does not distinguish between standard header files and user-defined files. The files are searched according to the SSV search list that you specified.
- When using an SSV search list with the native compilers, the default subvolume is searched. Therefore, you do not need to add an `SSV` pragma for the default subvolume in native mode. When using SSVs with the TNS compilers, however, you do need to add an `SSV` pragma for the default subvolume.
- For the default searching paradigm used with `#include` directives, see **#include** on page 182.
- The way that you specify a # `include` directive affects the operation of an SSV search list or a Directories list:

  - For the TNS C compiler, the `include` directive searches for the specified file (in double quotes) in the current default Guardian volume and subvolume. If the file is not found, the directive searches in the compiler's Guardian volume and subvolume (e.g.`$system.system`). If pragma SSV<*n*> is used, the Guardian subvolumes are then searched.
  - For the TNS/R C compiler, TNS/E C compiler, and the TNS/X C compiler, the `include` directive searches for the specified file (in double quotes) in the Guardian volume and subvolume or OSS directory containing the source file. If the file is not found, the directive searches in the compiler's Guardian volume and subvolume or OSS directory (e.g.`$system.system` on Guardian

and /usr/include on OSS). If pragma SSV<*n*> or thec89-I flag is used, the Guardian subvolumes or OSS directories are then searched.

- SSV pragmas on the command line take precedence over SSV pragmas in source files; that is, those in source files are appended to those from the command line. If no SSV pragma appears on the command line, two are implicitly created (without user notification) for the current subvolume and compiler subvolume. This gives those locations precedence over any SSV pragmas in source files, including CPATHEQ files. To override the default SSV pragmas, specify at least one SSV pragma on the command line.

Examples

1. This example specifies three search subvolumes:

```
c / in testc, out $s.#xxx / obj;run,ssv0 "$a.b", ssv1 "$b.d",
ssv2 "$system.system"
```

2. This example specifies a search subvolume ($A.B) and a search volume ($C):

```
#pragma ssv0 "$a.b", ssv1 "$c"
```

STDFILES

The STDFILES pragma controls the automatic opening of the three standard files: stdin, stdout, and stderr. The STDFILES pragma allows the C library to automatically open the three standard files. The NOSTDFILES pragma suppresses the automatic opening of these files.

```
[NO]STDFILES
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	STDFILES	N.A.
G-series TNS c89 utility	STDFILES	N.A.
TNS/R native C and C++ compilers	STDFILES	N.A.
Native c89 and c99 utilities	STDFILES	N.A.

Table Continued

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS/E native C and C++ compilers	STDFILES	N.A.
TNS/X native C and C++ compilers	STDFILES	N.A.

Usage Guidelines

- For the native compilers, the STDFILES pragma can be entered only on the compiler RUN command line. For the TNS compilers, the pragma can be specified with the -W[no]stdfiles flag of the c89 or the c99 utility.
- The C compiler defaults to STDFILES if pragma SYSTYPE GUARDIAN is specified. You cannot specify the STDFILES pragma if pragma SYSTYPE OSS is specified. The compiler issues a warning.
- The STDFILES pragma is effective only for compiling the module that contains the main function. These pragmas have no meaning for other modules, and the compiler issues a warning if the pragmas are used with other modules.
- In a mixed-language program, the three standard files are not automatically opened unless the module containing the main function is written in the C programming language. If you need to open any of the three standard files for I/O operations in the C language, explicitly open each file by calling the fopen_std_file() function.

STRICT

The STRICT pragma directs the TNS C compiler or Cfront to generate a warning if it encounters one of a number of valid, but questionable, syntactic or semantic constructs.

```
STRICT
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	Not set	Not set
G-series TNS c89 utility	Not set	Not set
TNS/R native C and C++ compilers	N.A.	N.A.

Table Continued

	SYSTYPE GUARDIAN	SYSTYPE OSS

Native
`c89`
and
`c99`
utilities
	N.A.	N.A.

TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	N.A.	N.A.

Usage Guidelines

- The `STRICT` pragma can be entered on the compiler RUN command line or in the source text.
- The `STRICT` pragma causes these additional warning messages to be issued: 5, 36, 84, 92, 93, 94, 96, 98, 105, 107, 108, 119, 148, 151, 179, 217, and 218. For information about messages, see **TNS C Compiler Messages** on page 438.
- The constructs that the `STRICT` pragma causes the compiler to diagnose include:
  - Function declarations and definitions that do not use function-prototype syntax
  - Calls to functions that do not have a function-prototype declaration or definition in scope
  - Calls to functions that have no nonprototype declaration or definition in scope
  - Calls to library routines that occur before inclusion of the library header that declares or defines the routine
  - Object declarations that do not include a type specifier
  - Nonprototype function definitions that do not specify the types of all parameters
  - A type conversion that might cause loss of data or precision without an explicit cast expression
  - Access of the value of a local variable before it appears to have been initialized
  - A constant used as the controlling condition of a `do`, `for`, `if`, `switch`, or `while` statement or of a conditional expression (using the `?:` operator)
- The native C and C++ compilers do not support this pragma. Native compilers perform strict syntactic and semantic checking by default.

SUPPRESS

The `SUPPRESS` pragma controls the generation of compiler-listing text, regardless of the status of the `LIST` pragma.

```
[NO]SUPPRESS
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	NOSUPPRESS	NOSUPPRESS
G-series TNS c89 utility	SUPPRESS	SUPPRESS
TNS/R native C and C++ compilers	NOSUPPRESS	NOSUPPRESS
Native c89 and c99 utilities	SUPPRESS	SUPPRESS
TNS/E native C and C++ compilers	NOSUPPRESS	NOSUPPRESS
TNS/X native C and C++ compilers	NOSUPPRESS	NOSUPPRESS

Usage Guidelines

- The SUPPRESS pragma can be entered only on the compiler RUN command line. The pragma can be specified with the -W[no]suppress flag of the c89 or the c99 utility.
- The SUPPRESS pragma overrides the ICODE, INNERLIST,LIST,LMAP,MAP, and PAGE pragmas.

SUPPRESS_VTBL

The SUPPRESS_VTBL command-line option suppresses the definition of virtual function tables in cases where the heuristic used by the compiler to decide on definition of virtual function tables provides no guidance.

The virtual function table for a class is defined in a compilation if the compilation contains a definition of the first non-inline, nonpure virtual function of the class. For classes that contain no such function, the default behavior is to make the definition a local static entity. This option is valid only for native TNS/R C+ + only.

```
SUPPRESS_VTBL
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS c89 utility	N.A.	N.A.
TNS/R native C and C++ compilers	Not set	Not set
Native c89 and c99 utilities	Not set	Not set
TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

- SUPPRESS_VTBL can either be entered on the compiler RUN command line (CPPCOMP or NMCPLUS) or be specified with the -Wsuppress_vtbl option of the c89 or the c99 utility.
- See also **FORCE_VTBL** on page 236.

SYMBOLS

The SYMBOLS pragma controls the inclusion of symbol information in the object file for use by a symbolic debugger. The SYMBOLS pragma directs the compiler to include symbol information in the object file, and NOSYMBOLS directs the compiler to omit this information.

```
[NO]SYMBOLS [ ( NODEFINES ) ]
```

(NODEFINES)

directs the compiler to omit symbol information regarding preprocessor symbols and macros. Omitting this information can dramatically reduce the size of a macro-intensive program’s object file. This option is not valid for native mode.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	NOSYMBOLS	NOSYMBOLS
G-series TNS c89 utility	NOSYMBOLS	NOSYMBOLS
TNS/R native C and C++ compilers	NOSYMBOLS	NOSYMBOLS
Native c89 utility	NOSYMBOLS	NOSYMBOLS
c99 utility	N.A.	N.A.
TNS/E native C and C++ compilers	NOSYMBOLS	NOSYMBOLS
TNS/X native C and C++ compilers	NOSYMBOLS	NOSYMBOLS

Usage Guidelines

- For native C and C++ programs, the SYMBOLS pragma can be entered only on the compiler RUN command line or be specified with the -g option of the c89 utility. For TNS C and C++ programs, the SYMBOLS pragma must appear on the compiler RUN command line or at the start of the source text before any declarations or source code statements.
- The SYMBOLS pragma affects the INSPECT pragmas: if you specifySYMBOLS, the compiler automatically enables Visual Inspect and the source-level symbolic debugger, as though you had explicitly specified INSPECT.
- In native mode, the SYMBOLS pragma cannot include the NODEFINES option.

SYNTAX

The SYNTAX pragma directs the compiler to not generate an object file but merely to check the source text for syntactic and semantic errors.

```
SYNTAX
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	Not set	Not set
G-series TNS `c89` utility	Not set	Not set
TNS/R native C and C++ compilers	Not set	Not set

Native
`c89`
and
`c99`
utilities
	Not set	Not set

TNS/E native C and C++ compilers	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

- For TNS C and C++ programs, the `SYNTAX` pragma must appear on the compiler RUN command line or at the start of the source text before any declarations or source code statements.
- For native C and C++ programs, the `SYNTAX` pragma can be entered on the compiler RUN command line or be specified with the `-Wsyntax` flag of the `c89` or the `c99` utility.

SYSTYPE

The `SYSTYPE` pragma controls whether the generated code's target execution environment is the NonStop environment.

```
SYSTYPE { GUARDIAN | OSS }
```

The pragma default settings are:

	Guardian Environment	OSS Environment
TNS C compiler	GUARDIAN	N.A.
G-series TNS `c89` utility	N.A.	OSS

Table Continued

	Guardian Environment	OSS Environment
TNS/R native C and C++ compilers	GUARDIAN	N.A.
Native c89 and c99 utilities	N.A.	OSS
TNS/E native C and C++ compilers	GUARDIAN	N.A.
TNS/X native C and C++ compilers	GUARDIAN	N.A.

Usage Guidelines

- The SYSTYPE pragma can be entered on the compiler RUN command line or be specified with the –Wsystype flag of the c89 or the c99 utility. A SYSTYPE pragma in the source text does not change the target environment; it is only an affirmation of the target environment.
- The compiler issues an error if a SYSTYPE pragma specified in the source text differs from:
  - The SYSTYPE pragma specified on the RUN command line.
  - The -Wsystype flag of the c89 or the c99 utility.
  - The compiler default environment.
- If you run the compiler in the Guardian environment, the default setting is SYSTYPE GUARDIAN. If you run the compiler in the OSS environment (with the c89 or the c99 utility), the default setting is SYSTYPE OSS.
- The SYSTYPE GUARDIAN and SYSTYPE OSS pragmas set the default values for many other pragmas.
- Pragma SYSTYPE GUARDIAN defines the _GUARDIAN_TARGET macro. Pragma SYSTYPE OSS defines the _OSS_TARGET and _XOPEN_SOURCE macros. Because the pragma defines and undefines macros while compiling, you must specify the SYSTYPE SYSTYPE pragma before any definitions in the RUN command line or in a source file.
- If SYSTYPE GUARDIAN or SYSTYPE OSS is specified with RUNNABLE option or -c is not specified with -Wsystype(c89, c99, c11), the compiler passes either -set SYSTYPE OSS or -set SYSTYPE GUARDIAN to the linker.
- The SYSTYPE pragma has no meaning for a DLL. A DLL may be usable by either a Guardian process or an OSS process, or both, depending on how various parts of the DLL were written and compiled. You must have information about how the DLL was written and compiled to use it appropriately.

TANDEM_FLOAT

The TANDEM_FLOAT directive specifies that the native C or C++ compiler is to use the proprietary Tandem floating-point format for performing floating-point operations.

```
TANDEM_FLOAT
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS c89 utility	N.A.	N.A.
TNS/R native C and C++ compilers	TANDEM_FLOAT	TANDEM_FLOAT
Native c89 utility	For TNS/R code: TANDEM_FLOAT For TNS/E and TNS/X code: IEEE_FLOAT	For TNS/R code: TANDEM_FLOAT For TNS/E and TNS/X code: IEEE_FLOAT
c99 utility 1	N.A.	N.A.
TNS/E native C and C++ compilers	IEEE_FLOAT	IEEE_FLOAT
TNS/X native C and C++ compilers	IEEE_FLOAT	IEEE_FLOAT

[1] The c99 utility supports IEEE floating point only.

Usage Guidelines

- The TANDEM_FLOAT directive can be entered on the compiler RUN command line or with the –WTandem_float flag of the c89 utility.
- In RVUs preceding the G06.06 release, only the proprietary Tandem floating-point format was available. With the introduction of the processor-hosted IEEE floating-point format at G06.06, you can choose either Tandem floating-point format or IEEE floating-point format. For more details, see **Compiling and Linking Floating-Point Programs** on page 386.
- For a high-level discussion of differences between IEEE and Tandem floating-point, see **IEEE Floating-Point Arithmetic** on page 82.
- For more detailed information about differences between the IEEE and Tandem floating-point formats, see the pragma **IEEE_FLOAT** on page 248.

TARGET

The `TARGET` pragma is a command-line directive and directs the native compilers to create an output file for the specified target hardware platform.

```
TARGET value
```

where, value is:

```
{TNS/X | X86 | TNS/E | IPF }
```

`TNS/X | X86`

The output file is for the TNS/X platform. The compiler driver invokes the TNS/X targeted compiler components.

`TNS/E | IPF`

The output file is for the TNS/E platform. The compiler driver invokes the TNS/E targeted compiler components.

Usage Guidelines

- `TARGET` is a command-line directive that must be entered on the compiler RUN command line, not in the source text.
- `TARGET` is only supported in the TNS/X native C/C++ compiler.

TEMPEXTENT

The `TEMPEXTENT` pragma controls the size of the temporary files created by the compiler.

```
TEMPEXTENT < number>
```

The default TEMPEXTENT < *number*> is 192.

The valid values are in the range of 2 to 32767.

On compiling large programs, the compiler fails and returns the error message: `fprintf failed: not enough disk space`. When you encounter this error, adding the TEMPEXTENT < *number*> directive to the Guardian XCOBOL command resolves the issue. HPE recommends setting the < *number*> to 1024 to resolve this issue.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	N.A.	N.A.
Native `c89` and `c99` utilities	N.A.	N.A.

Table Continued

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	TEMPEXTENT	N.A.

THREAD

The THREAD pragma enables C++11 threads. C++11 threads are supported for L18.08 RVU onwards. By default, the thread support is disabled when using CPPCOMP to compile or link VERSION4 C++ programs. This pragma is supported for SYSTYPE OSS. The applications must link with 32-bit -XCPPTDLL or 64-bit -WCPPTDLL to use C++11 threads.

This option is only supported by CPPCOMP, it is not recognized by CCOMP. The default is "not set".

Compiling with CPPCOMP, use:

```
THREAD
```

Compiling with c11, use :

```
-Wthread
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS c89 utility	N.A.	N.A.
TNS/R native C and C++ compilers	N.A.	N.A.
Native c89 and c99 utilities	N.A.	N.A.
TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	N.A.	THREAD

TRIGRAPH

The TRIGRAPH pragma controls whether the TNS C compiler or Cfront translate trigraphs for the current compilation. When the TRIGRAPH pragma is in effect, the compiler recognizes the trigraphs and replaces them with their corresponding C tokens. When NOTRIGRAPH is in effect, the compiler does not support the trigraph feature and treats each character in the trigraph as a C token.

```
[NO]TRIGRAPH
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	NOTRIGRAPH	NOTRIGRAPH
G-series TNS c89 utility	NOTRIGRAPH	NOTRIGRAPH
TNS/R native C and C++ compilers	N.A.	N.A.
Native c89 and c99 utilities	N.A.	N.A.
TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	N.A.	N.A.

Usage Guidelines

- The TRIGRAPH pragma can be entered on the compiler RUN command line or in the source text.
- The native C and C++ compilers do not support this pragma. Native compilers always translate trigraphs.

VERSION1

The VERSION1 pragma is a command-line directive for TNS/R native mode C++ that instructs the C++ compiler to compile using the D40 version or dialect of C++, rather than a more recent version. This pragma is not accepted by the TNS, TNS/E, or TNS/X compilers.

```
VERSION1
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS c89 utility	N.A.	N.A.

Table Continued

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS/R native C and C++ compilers	VERSION3 for C++, N.A. for C	VERSION3 for C++, N.A. for C
Native c89 utility	VERSION3 for C++, N.A. for C	VERSION3 for C++, N.A. for C
c99 utility	N.A.	N.A.
TNS/E native C and C++ compilers	VERSION3 for C++, N.A. for C	VERSION3 for C++, N.A. for C
TNS/X native C and C++ compilers	VERSION3 for C++, N.A. for C	VERSION3 for C++, N.A. for C

Usage Guidelines

- As of G06.20, the default for native mode C++ compilation is VERSION3 (not VERSION1). If you are going to recompile an application that used the previous default (VERSION1), you must specify the VERSION1 pragma. See also **VERSION2** on page 324 and **VERSION3** on page 326.
- You can enter the VERSION1 directive on the compiler RUN command line when specifying NMCPLUS on Guardian environment, or using the -Wversion1 flag when specifying c89 on OSS environment. You cannot enter the VERSION1 directive in the source file.
- All modules of an application must be built using the same version of the Standard C++ Library. For example, you must compile all modules using the same version directive (VERSION1, VERSION2, or VERSION3). Mixing versions within an application can cause unpredictable results. Additionally, the linkers and the NonStop OS perform version checking. Attempting to mix VERSION3 with either of the other versions will yield an error (for link files) or a warning (for load files) from the linker, and a run-time error at load time from the NonStop OS.
- Using the VERSION1 directive with the D45 (or later) native C++ compiler produces an object file that is compatible with an object file produced by the D40 native C++ compiler but not compatible with an object file produced by the D45 (or later) native C++ compiler using the VERSION2 or VERSION3 directives.

  New C++ features introduced at the D45 RVU mean that objects compiled to take advantage of these features are incompatible with objects produced using earlier versions of the compiler.

  Moreover, object files produced by the D45 C++ compiler using the default at D45 (VERSION1) are not binary compatible with object files produced by the D45 C++ compiler using the VERSION2 directive or with object files produced by the G06.20 C++ compiler using the default, VERSION3.
- The nld linker issues an error or warning if you attempt to link several C++ modules that were compiled with different version directives. You can use the eld, xld, or ld linker only with VERSION2 or VERSION3.
- VERSION1 does not support the IEEE floating-point format for performing floating-point arithmetic.
- VERSION1 does not support Position-Independent Code (PIC) or the use of DLLs.

- In the G06.20 RVU, all the native C++ header files have been combined into one product number (T2824). These headers have been modified to identify the version used in the compile and to redirect calls to the correct library.
- If you include the RUNNABLEoption when compiling VERSION1 on Guardian environment (or if you do not include -c On OSS environment), the compiler automatically links:
  - ZCPLGSRL (Guardian C++ library) or ZCPLOSRL(OSS C++ library)
  - LIBCOBEY for a TNS/R program (an OBEY file that links the C run-time library and the Common Run-Time Environment [CRE])
- VERSION1supports version 6.1 of Tools.h++.
- To use the Tools.h++ library (version 6.1) when using VERSION1, you need to link ZTLHGSRL (the Guardian Tools.h++ library) or ZTLHOSRL (the OSS Tools.h++ library). If you are compiling a loadfile (using RUNNABLE in Guardian), you can link to Tools.h++ by specifying the correct SRL in a LINKFILE pragma or by using the -Wnld_obey option to c89in OSS. For more details about SRLs, see **Shared Run-Time Libraries (SRLs)** on page 377.

VERSION2

The VERSION2 pragma is a command-line directive for native mode C++ that instructs the C++ compiler to compile using the dialect or features available beginning with the D45 version of the HPE C++ language. This pragma is not accepted by the TNS compilers.

```
VERSION2
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS c89 utility	N.A.	N.A.
TNS/R native C and C++ compilers	VERSION3 for C++, N.A. for C	VERSION3 for C++, N.A. for C
Native c89 utility	VERSION3 for C++, N.A. for C	VERSION3 for C++, N.A. for C
c99 utility	N.A.	N.A.

Table Continued

	SYSTYPE GUARDIAN	SYSTYPE OSS

TNS/E native C and C++ compilers	`VERSION3`
	for C++, N.A. for C
		`VERSION3`
		for C++, N.A. for C

TNS/X native C and C++ compilers	`VERSION3`
	for C++, N.A. for C
		`VERSION3`
		for C++, N.A. for C

Usage Guidelines

- You can enter the `VERSION2` directive on the compiler RUN command line when specifying NMCPLUS or CPPCOMP on Guardian environment, or using the `-Wversion2` flag of the `c89` utility. You cannot enter the `VERSION2` directive in the source file.
- `VERSION2` supports DLLs (Dynamic-Link Libraries) and Position-Independent Code (PIC).
- Using the `VERSION2` directive enables you to use these language features of the native C++ compiler:
    - Exception handling
    - Namespaces
    - The bool type
    - The wchar_t type
    - Array new and delete
    - Run-time type identification (RTTI)
    - New-style casts (`static_cast, reinterpret_cast`, and `const_cast`)
    - Explicit instantiation of templates
    - Support of partial specialization of templates
    - Support of `extern` inline functions
    - enum types are considered to be nonintegral types

      These language features are part of the proposed C++ standard introduced by the *Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++*, 2 December 1996, X3J6/96-0225 WG21/N1043.

      If you want to use any of these `VERSION2` features in an application originally compiled with any pre-D45 RVU of the TNS/R native C++ compiler, you must recompile and relink all modules of the program using the `VERSION2`directive.

- Object files compiled using `VERSION2` take advantage of the features available beginning at the D45 RVU and are incompatible with object files compiled using earlier or later versions of the native C++ compiler.
    - If you compile an application composed of multiple modules, you must compile all modules using the same version directive.

For example, if you use the VERSION2 directive to compile any module of a program, you must compile all modules of the program using the VERSION2directive.

  - The linkers issue an error or warning if you attempt to link several C++ modules that were compiled with different version directives.
  - You can disable version checking by designating a loadfile CPPNEUTRAL using a linker utility. For more details, see the *eld and xld Manual*, the *ld Manual*, or the *nld Manual*.

- You must use the VERSION2 command-line directive to use these middleware products:
  - Version 2 of the Standard C++ Library. See **Using the Standard C++ Library** on page 88.
  - Version 7 of Tools.h++. See **Accessing Middleware Using HPE C and C++ for NonStop Systems** on page 104.
- VERSION2 supports IEEE floating-point format.
- supports Version 7 of Tools.h++ and version 2 of the Standard C++ Library.)
- In the G06.20 RVU, all the native C++ header files were combined into one product number. These headers were modified to identify the version used in the compile and to redirect calls to the correct library.
- If you include the RUNNABLE option when compiling VERSION2 on Guardian environment (or if you do not include -c On OSS environment), the compiler automatically links:
  - ZCPLSRL(C++ run-time library)
  - (ZRWSLSRLStandard C++ Library, VERSION2)
  - LIBCOBEY for a TNS/R program (an OBEY file that links the C run-time library and the Common Run-Time Environment [CRE])
  - CPPINIT (non-PIC), or CPPINIT2 for CALL_SHARED (TNS/R PIC) (allows you to override the new and delete functions)
- ZCPPCDLL and ZCPP2DLL (common C++ run-time library and VERSION2-specific C++ standard library) must be linked in for TNS/E programs by the user. XCPPCDLL and XCPP2DLL (common C++ run-time library and VERSION2-specific C++ standard library) must be linked in for TNS/X programs by the user.

  For more details about the TNS/R SRLs, see **Shared Run-Time Libraries (SRLs)** on page 377 . For more details about the TNS/E or TNS/X DLLs, see **Dynamic-Link Libraries (DLLs)** on page 393 .

VERSION3

The VERSION3 pragma is a command-line directive for native mode C++ that instructs the compiler to use the features of the G06.20 dialect of the C++ language (VERSION3). This pragma is not accepted by the TNS compilers.

```
VERSION3
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS c89 utility	N.A.	N.A.
TNS/R native C and C++ compilers	VERSION3 for C++ N.A. for C	VERSION3 for C++ N.A. for C
Native c89 utility	VERSION3 for C++ N.A. for C	VERSION3 for C++ N.A. for C
c99 utility	VERSION3	VERSION3
TNS/E native C and C++ compilers	VERSION3 for C++ N.A. for C	VERSION3 for C++ N.A. for C
TNS/X native C and C++ compilers	VERSION3 for C++ N.A. for C	VERSION3 for C++ N.A. for C

Usage Guidelines

- You can enter the VERSION3 directive on the compiler RUN command line when specifying CPPCOMP or NMCPLUS on Guardian environment, or using the -Wversion3 flag of the c89 or the c99 utility in the OSS environment. You cannot enter the VERSION3 directive in the source file.
- VERSION3 is the default version of the native C++ compiler for the G06.20 RVU and for subsequent RVUs until a new version of the compiler is released.
- If you specify no version directive (or if you include the VERSION3 directive) with the G06.20 native C++ compiler, you will produce a binary that is compatible with binaries produced by the G06.20 compiler but not the D45 compiler (VERSION2) or any earlier compiler.
- The VERSION3 directive specifies the use of the Standard C++ Library ISO/IEC version 3, and the C++ Standard headers. VERSION3 enforces the ISO/IEC IS 14882:1998 standard for C++. The ISO C++ standard is identical to the ANSI C++ standard. The VERSION3 Standard C++ Library ISO/IEC is combined with the C++ run-time library into one public SRL (named ZSTLSRL) on systems running G-series RVUs but is separated into three component libraries (ZCPPCDLL, ZCPP2DLL, and ZCPP3DLL) on systems running H-series RVUs or J-series RVUs and (XCPPCDLL, XCPP2DLL, and XCPP3DLL) on systems running L-series RVUs to implement the BUILD_NEUTRAL_LIBRARY pragma. For more details about theVERSION3 Standard C++ Library, see the *Standard C++ Library Reference ISO/IEC (VERSION3)* on the product documentation CD, the HPE NonStop Technical Library (NTL).
- VERSION3 supports IEEE floating-point format.
- VERSION3 supports DLLs (Dynamic-Link Libraries) and Position-Independent Code (PIC).
- VERSION3 does not support any version of the Tools.h++ library.

- All modules of an application must be built using the same version of the Standard C++ Library. For example, you must compile all modules using the same version directive (VERSION1, VERSION2, or VERSION3). Mixing versions within an application can cause unpredictable results.

  Beginning at G06.20, the native linkers enforce non-mixing of versions. The `ld` and `nld` linkers issue a warning if you attempt to link modules compiled with `VERSION1` and `VERSION2` . An error is issued by `nld` if `VERSION3` is mixed with either `VERSION1` or `VERSION2`. In addition, `eld`, `xld`, and `ld` issue an error on any `VERSION1` linkfile in addition to any mixing of `VERSION2` with `VERSION3`.

  You can build a neutral C++ dialect DLL using the `BUILD_NEUTRAL_LIBRARY` pragma.

  You can disable version checking by designating a loadfile `CPPNEUTRAL` using a linker. You can also display the C++ version of a loadfile by using the `LISTATTRIBUTES` command of `enoftxnoft`, or `noft`. For more details, see the:

  - *eld and xld Manual*
  - *enoft Manual*
  - *xnoft Manual*
  - *ld Manual*
  - *nld Manual*
  - *noft Manual*

- Before compiling existing applications using `VERSION3`, you should first use the **MIGRATION_CHECK** on page 268 directive along with the `VERSION2` pragma to examine the code for references to functions in changed or unsupported headers.
- The `VERSION3` Standard C++ Library supports wide characters as described in the standard, and converts from Multi-Byte to wide.
- At the G06.20 RVU, all the native C++ header files were combined into one product number (T2824). These headers were modified to identify the version used in the compile and to redirect calls to the correct library.
- `VERSION3` headers do not typically have a `.h` file extension. For example, you should specify:

  ```
  #include <iostream>
  ```

  instead of: `#include <iostream.h>` (OSS) or instead of: `#include <iostreah>` (Guardian)

  For `VERSION3` you can use the standard header names without truncation as long as you also use the CPATHEQ file named SLMAP as described in **Pragmas for the Standard C++ Library** on page 101.

- The T2824 C++ headers include a few that end with `.h` (such as `fstream.h` and `iomanip.h`) for compatibility with previous C++ versions. If you include one of these `.h` header files when using `VERSION3`, the compiler issues this error:

  ```
  This header file is incompatible with C++ version 3
  ```

- HPE recommends that `VERSION3` users use the C-name headers that are part of the Standard C++ Library instead of the older C headers. For example, specify:

  ```
  #include <cstdio>
  ```

instead of:

```
#include <stdio.h>
```

- If you include the RUNNABLE option when compiling VERSION3 on Guardian environment (or if you do not include -c On OSS environment), the compiler automatically links:
  - ZSTLSRL for TNS/R programs, ZCPPCDLL and ZCPP3DLL for TNS/E programs, or XCPPCDLL and XCPP3DLL for TNS/X programs (the Standard C++ Library and the C++ run-time library). These are for 32-bit programs. For 64-bit programs, the compiler links YCPPCDLL and YCPP3DLL for TNS/E programs, or WCPPCDLL and WCPP3DLL for TNS/X programs.
  - LIBCOBEY for a TNS/R program (an OBEY file that links the C run-time library and the Common Run-Time Environment [CRE])
  - CPPINIT3 for a non-PIC code, or
  - CPPINIT4 for a TNS/R PIC (lets you override the new and delete functions)

    For information about TNS/R SRLs, see **Shared Run-Time Libraries (SRLs)** on page 377. For more details about the TNS/E or TNS/X DLLs, see **Dynamic-Link Libraries (DLLs)** on page 393.

VERSION4

The VERSION4 pragma is a command-line directive for native mode C++ that instructs the compiler to compile the source using the 2011 C++ Standard (ISO/IEC 14882:2011). This pragma is available only on TNS/X systems.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS c89 utility	N.A.	N.A.
TNS/R native C and C++ compilers	VERSION3 for C++ N.A. for C	VERSION3 for C++ N.A. for C
Native c89 utility	VERSION3 for C++ N.A. for C	VERSION3 for C++ N.A. for C
Native c99 utility	VERSION3 for C++ N.A. for C	VERSION3 for C++ N.A. for C
Native c11 utility	VERSION4 for C++ N.A. for C	VERSION4 for C++ N.A. for C
TNS/E native C and C++ compilers	VERSION3 for C++ N.A. for C	VERSION3 for C++ N.A. for C
TNS/X native C and C++ compilers	VERSION3 for C++ N.A. for C	VERSION3 for C++ N.A. for C

Usage Guidelines

- You can enter VERSION4 directive on the compiler RUN command line when specifying CPPCOMP in the Guardian environment, or using the –Wversion4 flag of the c11 utility in the OSS environment. You cannot enter the VERSION4 directive in the source file.
- All modules of an application must be built using the same version of the standard C++ library.

WARN

The WARN pragma controls the generation of all or selected warning messages. The WARN pragma enables the compiler to generate all or a selected set of warning messages, and NOWARN disables the compiler from generating all or a selected set of warning messages.

```
[NO]WARN [ warning-list ]

warning-list:
   ( warning-number [, warning-number ]... )
```

warning-list

is a parenthesized, comma-separated list of warning-message numbers. This list represents the warning messages whose generation the compiler is to enable or disable.

If you omit *warning-list*, the WARN or NOWARN pragma affects all warning messages.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	WARN	WARN
G-series TNS c89 utility	WARN	WARN
TNS/R native C and C++ compilers	WARN	WARN
Native c89 and c99 utilities	WARN	WARN
TNS/E native C and C++ compilers	WARN	WARN
TNS/X native C and C++ compilers	WARN	WARN

Usage Guidelines

- The WARN pragma can be entered on the compiler RUN command line or in the source text. The pragma can also be specified with the -W[no]warn flag of the c89 or the c99 utility.
- There is no correspondence between the warning-message text or numbers between the TNS C compiler and the native C and C++ compilers.

Examples

1. In this example, the compiler generates only warning messages 153 and 157.

```
#pragma nowarn
#pragma warn (153,157)
```

2. In this example, the compiler generates all warning messages except warning 153.

```
#pragma nowarn (153)
```

WIDE

The WIDE pragma specifies the data model, which defines the size of the data type int. The WIDE pragma specifies the 32-bit data model. NOWIDE specifies the 16-bit data model. (The 32-bit data model is also referred to as the wide-data model.)

```
[NO]WIDE
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	Not set	WIDE
G-series TNS c89 utility	WIDE	WIDE
TNS/R native C and C++ compilers	N.A	N.A.
Native c89 and c99 utilities	N.A	N.A
TNS/E native C and C++ compilers	N.A	N.A
TNS/X native C and C++ compilers	N.A	N.A

Usage Guidelines

- The `WIDE` pragma can be entered on the compiler RUN command line or in the source text.
- The TNS C++ translator, Cfront, is `WIDE` by default. The TNS C compiler is effectively `NOWIDE` by default. To compile TNS C++ programs for the Guardian environment that use the 32-bit data model, specify pragma `WIDE` in the C++ source file or on the TNS C++ preprocessor, Cprep, run command line.
- In source text, the `WIDE` pragma must be specified before any declarations or source code statements.
- The native C and C++ compilers do not support these pragmas. Native compilers generate only programs that use the 32-bit data model.
- The `WIDE` pragma compiles only under the large-memory model. The combination of the `WIDE` and `NOXMEM` pragmas is diagnosed as an error.
- All translation units of a program must specify the same data model. Programs composed of Guardian modules (modules compiled for the Guardian environment) and OSS modules (modules compiled for the OSS environment) must use the 32-bit data model.

  For more information about the two data models, see **Two Data Models: 16-Bit and ILP32** on page 434.

- The preprocessor variable `__INT32` is defined when the `WIDE` pragma is in effect.

XLD(arg)

The `XLD` pragma specifies arguments to be passed to the `xld` utility, the TNS/X linker for PIC (Position-Independent Code).

```
XLD(arg)
```

arg

is any argument accepted by the `xld` utility.

For more details on valid syntax and semantics, see the *eld and xld Manual*.

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	N.A.	N.A.
G-series TNS `c89` utility	N.A.	N.A.
TNS/R native C and C++ compilers	N.A.	N.A.

Table Continued

	SYSTYPE GUARDIAN	SYSTYPE OSS
Native c89 and c99 utilities	Not set	Not set
TNS/X native C and C++ compilers	Not set	Not set

Usage Guidelines

- The XLD pragma is a command-line directive and must not be entered in the source text.
- On Guardian environment, the XLD pragma must be entered on the compiler RUN command line for TNS/X native C/C++. On OSS environment, specify the XLD pragma by using the `-Wxld=`*`arg`* flag for the c89 or the c99 utility.
- The XLD pragma does not actually invoke the xld linker. To invoke xld, you must include other pragmas such as **RUNNABLE** on page 292, **SHARED** on page 300, or **LINKFILE** on page 262. If xld is not invoked, this pragma is ignored.
- This TNS/X native C command example shows the XLD pragma. Note that if you specify SHARED, no need to include RUNNABLE:

```
ccomp /in prog1/ prog1o; runnable, xld(-set floattype neutral)
```

XMEM

The XMEM pragma controls which memory model, large or small, the object file uses. The XMEM pragma specifies the large-memory model. NOXMEM specifies the small-memory model.

```
[NO]XMEM
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	XMEM	XMEM
TNS c89 utility	XMEM	XMEM
G-series TNS c89 utility	N.A	N.A

Table Continued

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS/R native C and C++ compilers	N.A	N.A
Native c89 and c99 utilities	N.A.	N.A.

Usage Guidelines

- The XMEM pragma can be entered on the compiler RUN command line or at the start of the source text before any declarations or source code statements.
- The native C and C++ compilers do not support the XMEM pragma. Native compilers generate only programs that use the large-memory model.
- TheNOXMEM pragma cannot be used if the pragma SYSTYPE OSS is specified. The compiler issues a warning.
- The NOXMEM pragma compiles only under the 16-bit data model. The combination of the WIDE and NOXMEM pragmas is diagnosed as an error.
- All translation units of a program must specify the same memory model. For more information about the two memory models, see **Two Memory Models: Large and Small** on page 431.
- The preprocessor variable __XMEM is defined when the XMEM pragma is in effect.

XVAR

The XVAR pragma controls whether the TNS C compiler places subsequent global or static aggregates in extended memory or in the user data segment. The XVAR pragma causes the TNS C compiler to allocate space in extended memory for any global or static aggregates that follow the pragma. NOXVAR causes the compiler to allocate space in the user data segment for any subsequent global or static aggregates.

The default for the large-memory model and the 32-bit or wide-data model is XVAR. To place a single global or static aggregate in the user data segment, precede it by #pragma NOXVAR and follow it by #pragma XVAR.

```
[NO]XVAR
```

The pragma default settings are:

	SYSTYPE GUARDIAN	SYSTYPE OSS
TNS C compiler	XVAR	XVAR
G-series TNS c89 utility	XVAR	XVAR
TNS/R native C and C++ compilers	N.A.	N.A.
Native c89 and c99 utilities	N.A.	N.A.
TNS/E native C and C++ compilers	N.A.	N.A.
TNS/X native C and C++ compilers	N.A	N.A

Usage Guidelines

- The XVAR pragma can be entered on the compiler RUN command line or in the source text.
- The TNS c89 utility does not support NOXVAR for Guardian or OSS modules.
- The XVAR pragma has no affect unless the XMEM or WIDE pragma is also in effect.
- You can use XVAR selectively, turning it on or off around declarations of different variables, but you must apply it consistently for each declaration of the same variable.
- Buffers used in certain Guardian procedure calls must be forced into the user data segment by use of NOXVAR so that they will be 16-bit addressable.
- All translation units of a program must specify the same data model. For more information, see the discussion of memory models in **Two Memory Models: Large and Small** on page 431.
- The native C and C++ compilers do not support the XVAR pragma. The native process memory architecture makes them unnecessary.

NOTE: For more information about using the cross compilers to compile programs with embedded SQL, see *HPE NonStop SQL/MX Release 3.2 Programming Manual for C and COBOL*.

Compiling, Binding, and Accelerating TNS C Programs

The TNS C compiler takes as input a module (a translation unit) and generates an object file. A module is a source file and all the headers and source files it includes, except for any source lines skipped as the result of conditional preprocessor directives.

The Binder collects and links object files generated by the C compiler and produces an executable object file (a program file).

The Accelerator processes TNS object files (object files generated by the C compiler and Binder) into accelerated object files for TNS/R systems. Accelerated object files run faster than TNS object files on TNS/R systems.

The Object Code Accelerator (OCA) processes TNS object files (object files generated by the C compiler and Binder) into accelerated object files for TNS/E systems. Accelerated TNS object files run faster than plain TNS object files on TNS/E systems. The Object Code Accelerator for TNS/X (OCAX) processes TNS object files into accelerated object files for TNS/X systems.

The SQL compiler processes TNS and accelerated object files and generates code for embedded SQL statements.

The G-series TNS `c89` utility controls the C compilation system in the Open System Services (OSS) environment. `c89` provides a simple interface to the components of the C compilation system, including the C language preprocessor, C compiler, Binder, Accelerator, and the SQL compiler.

There are two TNS C compilers. One compiler runs in the Guardian environment, and the other compiler runs in the OSS environment. Each compiler compiles Guardian and OSS programs and produces identical code. However, each compiler has different default pragma settings.

Selecting a Development Platform

A development platform consists of the hardware system and software environment available to compile, bind, accelerate, and run a program.

HPE NonStop TNS systems support only the Guardian environment. HPE NonStop native systems support both the Guardian and Open System Services (OSS) environments.

You can develop OSS TNS programs regardless of whether the OSS environment is available on the system. However, you cannot run and test OSS TNS programs on a system without the OSS environment, and you cannot run and test TNS or accelerated processes on a TNS/E or a TNS/X system in the OSS environment.

It is easier to develop a program in the environment in which it runs, but you can develop a program in one environment that runs in the other environment, with a few restrictions.

Development Platform Capabilities (TNS Programs) table describes the capabilities of each development platform.

Table 31: Development Platform Capabilities (TNS Programs)

Capability	TNS system or native system with Guardian environment	Native system with Guardian and OSS environments
Use Guardian development tools for Guardian TNS programs?	Yes	Yes
Use Guardian development tools for OSS TNS programs?	Yes	Yes
Use OSS development tools for Guardian TNS programs?	No	Yes (G-series only)
Use OSS development tools for OSS TNS programs?	No	Yes (G-series only)
Run Guardian TNS programs?	Yes	Yes
Run OSS TNS programs?	No	Yes (G-series only)

These restrictions apply to developing Guardian TNS programs with OSS tools:

- The TNS versions of OSS tools are not available on TNS/E or TNS/X systems.
- You cannot use the `NOWIDE`, `NOXMEM`, and `NOXVAR` pragmas. Therefore, you can develop programs that use only the 32-bit or wide data model.
- You cannot use the `RUNNABLE` and `SEARCH` pragmas. However, you can direct the TNS `c89` utility to bind implicitly after a compilation. You can also specify library files to be searched using the TNS `c89 -L` flag.
- You cannot use the `SSV` pragma. To specify search directories, use the TNS `c89 -I` flag.

Specifying Header Files

The macros and functions in the C run-time library are declared in header files. Each header file contains declarations for a related set of library functions and macros, in addition to variables and types that complete that set. If you use a function in the library, you should include the header file in which it is declared. You should not declare the routine yourself, because the declarations in the header files have provisions for several situations that can affect the form of a given declaration, including:

- Whether the routine is implemented internally as a function or a macro
- Whether the function is written in a language other than C
- Whether you are compiling for the small-memory or large-memory model
- Whether you are compiling for the 16-bit or 32-bit (wide) data model
- Whether you are compiling for the NonStop environment

In addition, the header file prototype declarations enable the C compiler to check parameters and arguments for compatibility, ensuring that function calls provide the correct number and type of arguments.

A single set of C library header files support both the Guardian and OSS environments. In the Guardian environment, header files are in $SYSTEM.SYSTEM and $SYSTEM.ZTCPIP by default. In the OSS environment, header files are in `/usr/include` and its subdirectories by default.

To specify header files, use the `#include` preprocessor directive. For more details and examples, see **Preprocessor Directives and Macros** on page 177.

In the Guardian environment, the `SSV` pragma specifies a search list of subvolumes for files specified in `#include` directives. For more details, see the description of pragma **SSV** on page 308.

In the OSS environment, the TNS `c89` utility `-I` flag specifies a search list of directories for files specified in `#include` directives.

The TNS `c89` utility is located in the `/nonnative/bin` directory on D40 and later releases; this utility is not available in H-series RVUs. By default, the native `c89` utility is run. You must set `/nonnative/bin` at the start of your `PATH` environment variable to get the TNS `c89` utility instead of the native `c89` utility.

For more details, see the TNS `c89` online reference page. To view this reference page, enter:

```
man -M /nonnative/usr/share/man c89
```

The `c89` reference page in D40 and later versions of the *Open System Services Shell and Utilities Reference Manual* describes the native `c89` utility.

While header files are optional (but strongly recommended) for programs that contain Guardian or OSS modules exclusively, header files are required for mixed-module programs. If you do not compile using header files, the Binder cannot correctly resolve external references to Guardian C and OSS functions.

Working in the Guardian Environment

In the Guardian environment, you can compile, bind, and accelerate C programs for either the Guardian or G-series Open System Services (OSS) environment. In the Guardian environment, you use the C compiler, the Binder (the BIND program), and the Accelerator (the AXCEL program) or OCA to develop your applications.

Compiling a C Module

The C compiler translates the source text of a module and produces these:

- An extensive compiler listing. Several pragmas enable you to control the content of this compiler listing.
- A nonexecutable object file, provided that the compiler encountered no errors during the compilation. If your C program comprises only a single module, use the `RUNNABLE` pragma to direct the compiler to produce a program file instead of a nonexecutable object file.

After compiling all the modules that compose your C program, use Binder to collect and combine them into a program file (an executable object file).

To compile a module, you start the C compiler process using the TACL command RUN as shown in this diagram:

```
[ RUN ] C / IN source [ , OUT listing ] [ , run-options ] /

   [ object ] [ ; compile-option [ , compile-option ]... ]

compile-option:
```

```
{ pragma                              }
{ define identifier [ integer-constant ] }
{ undefine identifier                    }
```

[RUN] C

is the TACL command to start the C compiler process. Note that the command keyword RUN is optional.

IN *source*

specifies the primary source file of the module.

OUT *listing*

specifies the file to which the C compiler writes the compiler listing. When specified, *listing* is usually a spooler location. If you omit the OUT option, the compiler writes the listing to your current default output file.

run-options

is a comma-separated list of additional options for the RUN command. These options are described in the *TACL Reference Manual*.

object

specifies the file to which the C compiler writes the object code for the source text. If you do not specify an object file, the compiler writes the object code to the file OBJECT in your current default volume and subvolume. If OBJECT cannot be created, the compiler writes the object code to the file ZZBI`nnnn` (where `nnnn` is a unique four-digit number) in your current default volume and subvolume.

compile-option

modifies compiler operation by specifying a compiler pragma or defining a preprocessor symbol.

pragma

is any compiler pragma. If you want, you can abbreviate the pragma name. The rule is that you must specify enough letters of the pragma name to make the name unique. For example, you need at least `RUNNAB` for the `RUNNABLE` pragma, because otherwise it would be confused with the `RUNNAMED` pragma.

define *identifier [integer-constant]*

defines *identifier* as a preprocessor symbol. If *identifier* is followed by an integer constant, it is defined as an object-like macro that expands to the given value. `define` is equivalent to using the `#define` preprocessor directive in source text.

undefine *identifier*

deletes *identifier* as a preprocessor symbol. Using undefine is equivalent to using the `#undef` preprocessor directive in source text.

Usage Guidelines

- The C compiler accesses source files as text-type logical files. Consequently, the source files you specify in a module must represent physical file types that the compiler can access as text-type logical files.
- The C compiler accepts logical source lines of up to 509 characters.
- The C compiler returns one of these completion codes:

0	The compilation completed successfully.
1	The compilation completed with warnings (but no errors).
2	The compilation completed with errors.
3	The compiler terminated abnormally as the result of an internal error.

Examples

1. This example directs the compiler to translate the source file ABSC (which contains an entire C program), send the compiler listing to the device $S.#ABSL, and store the program file under the name ABSO:

   ```
   C /IN absc, OUT $s.#absl/ abso; RUNNABLE
   ```

2. This example disables the run-time diagnostics that the assert function can issue by defining `NDEBUG` as a preprocessor symbol:

   ```
   C /IN appc, OUT $s.#appl/ appo; define NDEBUG
   ```

3. This examples directs the compiler to generate a TNS program that runs in the OSS environment:

   ```
   C /IN filec / fileo; SYSTYPE OSS
   ```

Binding a C Module

You must use Binder to collect and combine the object files into a program file (an executable object file) if:

- You do not use the `RUNNABLE` pragma when compiling a single-module program.

- Your program comprises several separately compiled modules.

To start Binder, enter the BIND command at the TACL prompt:

```
5> BIND
```

After starting Binder, you enter Binder commands to combine your object files and C run-time library object files to produce a program file. This diagram illustrates the Binder commands you can use to perform this task:

```
SELECT CHECK PARAMETER OFF
SET SYSTYPE environment
SET FILESYS file-system
ADD * FROM main-obj-file
ADD * FROM obj-file
SELECT SEARCH model-dependent-library-file
SELECT RUNNABLE OBJECT ON
SELECT LIST * OFF
SET HEAP value PAGES
BUILD program-file
```

SELECT CHECK PARAMETER OFF

disables checking of parameter number, type, and mode (value or reference) and of function return type across code blocks. If you do not disable these checks, Binder might generate several extraneous warning messages because C does not require functions to be declared before they are called.

SET SYSTYPE environment

specifies the execution environment for the program, either `GUARDIAN` for the Guardian environment or `OSS` for the G-series OSS environment. The default setting is `GUARDIAN`.

SET FILESYS *file-system*

specifies the file system used by Binder to resolve partially qualified file names, either `GUARDIAN` for the Guardian environment or `OSS` for the G-series OSS environment. If you specify `GUARDIAN`, Binder resolves file names with respect to the default volume and subvolume. If you specify `OSS`, Binder resolves file names with respect to the current working directory. The default setting is `GUARDIAN`.

ADD * FROM *main-obj-file*

adds the object file of your program's main module. The main module is the one that contains the definition of the function `main()`.

ADD * FROM *obj-file*

adds the object file of one of your program's other modules. You must repeat this command for each of your modules.

SELECT SEARCH *model-dependent-library-fil*e

directs Binder to search the specified model-dependent library file when resolving external references. You choose the model-dependent library file depending on the execution environment, the data model, and the memory model of your program. For more details on selecting the correct model-dependent library file, see **Specifying Library Files** on page 341.

The *model-dependent-library-file* file name must be fully qualified. The files are located in $SYSTEM.SYSTEM by default.

SELECT RUNNABLE OBJECT ON

directs Binder to create an executable object file (a program file) when it builds the object file.

SELECT LIST * OFF

disables generation of load maps and cross-reference listings.

SET HEAP value PAGES

specifies the heap size of the program.

BUILD program-file

directs Binder to build a program file using the files and options specified in the preceding commands. *program-file* is the name of the program file that Binder builds.

Specifying Library Files

To resolve references to external routines, Binder uses different model-dependent library files that depend on your program's execution environment, data model, and memory model. **Model-Dependent Library Files** table shows you how to select the correct library files for your programs.

Table 32: Model-Dependent Library Files

Execution Environment	Data Model	Memory Model	Library Files
Guardian module	16-bit	Small	CSMALL
Guardian module	16-bit	Large	CLARGE
Guardian module	32-bit	Large	CWIDE
Guardian module using fault-tolerant programming extensions	32-bit	Large	CWIDE and CNONSTOP
G-series OSS TNS module	32-bit	Large	COSS

The COSS and CWIDE library files support the 32-bit data model. Bind OSS modules with COSS and bind Guardian modules with CWIDE. If you compile all modules in a program using the header files supplied by HPE, you can bind using either COSS or CWIDE. If you do not use header files, you must bind using the library file that supports the module's environment.

You can only use static binding for Guardian programs. You can use static binding or dynamic binding for OSS programs. In static binding, the Binder resolves references to library functions by binding the functions into the program. In dynamic binding, the Binder resolves references to library functions using a TNS shared run-time library (SRL). Final resolution of references is performed at run time.

A TNS shared run-time library (SRL) is a special form of a TNS user library that can contain global variables. Programs that use an SRL cannot use a user library. HPE provides TNS SRLs for OSS and NonStop TUXEDO. You cannot build your own TNS SRLs. Do not confuse TNS SRLs with the TNS/R native SRLs provided on D40 and later software releases. For information on native SRLs, see **Shared Run-Time Libraries (SRLs)** on page 377.

To specify an SRL for an OSS TNS module compiled in the Guardian environment, do one of these:

- Move the compiled object code file to the G-series OSS environment and use the OSS environment TNS `c89` utility to specify the SRL and dynamically bind the program. For example, to specify an SRL for the object file `myobject`, enter:

  ```
  c89 -Wbind myobject
  ```

- Specify a Binder SET IMPORT command when building the program. For more details, see the *Binder Manual*.

For more details, see the TNS `c89` online reference page (`man -M /nonnative/usr/share/man c89`). The `c89` reference page in D40 and later versions of the *Open System Services Shell and Utilities Reference Manual* describes the native `c89` utility.

Changes in Binding CLIB

On C-series systems, the C run-time library resided in a file, CLIB, that had to be bound into each object file that contains C functions. On systems running D-series RVUs, the TNS C run-time library was split between two files, CLIB and CRELIB. Because CLIB and CRELIB were configured into the system library, you do not bind them into object files.

The D20 and later C run-time libraries have significant architectural differences. Because of these differences, D20 programs that bound in CLIB without also binding in CRELIB do not run correctly on D30 or later releases. You must bind such programs again without CLIB or CRELIB. CLIB is no longer released as a separate file. Check your BIND scripts to ensure that you do not bind in CLIB or CRELIB. Look for statements such as:

```
SELECT SEARCH clib
ADD * FROM clib
```

Delete any old copies of CLIB from $SYSTEM.SYSTEM.

Restrictions on Binding Caused by the ENV Pragma

Binder categorizes the ENV pragma options into three groups: OLD, NEUTRAL, and COMMON. For the most part, these groups match the various ENV options. **Binder Grouping of ENV Pragma Options** table shows how Binder classifies object files into one of three groups depending on the compiler version or specified ENV option.

Table 33: Binder Grouping of ENV Pragma Options

Binder Group	Generated by
OLD	C-series C compilers by default
COMMON	D-series C compilers by default D-series C compilers with ENV COMMON specified D-series C compilers with ENV LIBRARY specified
NEUTRAL	D-series C compilers with ENV EMBEDDED specified D-series C compilers with ENV LIBSPACE specified

These rules apply for binding modules together:

- You can bind object files that are in the same Binder group; the resulting object file runs in the same environment as the input object files.
- You can bind object files that include routines from both the OLD and NEUTRAL Binder groups; the resulting object file runs in a language-specific run-time environment.
- You can bind object files that include routines from both the COMMON and NEUTRAL Binder groups; the resulting object file runs in the CRE.
- You cannot bind object files that include routines from both the COMMON and OLD Binder groups.

When you bind object files compiled for different environments, each procedure retains its original ENV attribute. **Run-Time Environment Resulting From Binding Modules** table shows the run-time environment resulting from binding modules from different Binder groups together.

Table 34: Run-Time Environment Resulting From Binding Modules

	Binder Group		
Binder Group	OLD	COMMON	NEUTRAL
OLD	language-specific	Not allowed	language-specific
COMMON	Not allowed	CRE	CRE
NEUTRAL	language-specific	CRE	language-specific or CRE

Use the Binder INFO command with the DETAIL clause to show the ENV attribute of a particular data or code block. For more details, see the *Binder Manual*.

Accelerating C Programs

The Accelerator enables you to increase the performance of programs that run on TNS/R systems. The Object Code Accelerator (OCA) enables you to increase the performance of programs that run on TNS/E systems.

The Accelerator or OCA optimizes TNS programs to take advantage of the native architecture. Most TNS object code that has been accelerated runs faster on native systems than TNS object code that has not been accelerated. Most programs written for TNS systems do not require changes to run on native systems, either with or without acceleration.

The Accelerator and OCA take as input an executable TNS object file and produce as output an accelerated object file. The accelerated object file contains both the original TNS code and the logically equivalent optimized native instructions—accelerated object code.

The basic steps to accelerate a C program are:

1. Compile and bind the program.
2. Debug the program, if necessary.
3. Run the Accelerator. On TNS/R systems, you do this with the AXCEL command. On TNS/E systems, you use the OCA command. On TNS/X systems, you use the OCAX command.

   If the Accelerator, OCA, or OCAX issues an error message, correct the error and run the Accelerator, OCA, or OCAX again to produce an output file.
4. Run the NonStop SQL/MP compiler, if the program contains embedded NonStop SQL/MP statements.
5. Perform final testing on the accelerated program.

To save time in accelerating your programs, to produce the smallest possible accelerated object files, and to ensure that the Accelerator or OCA produces the most efficient code, to do these:

- Use C function prototypes for all your C routines.
- Generate and retain the Binder and Inspect symbols regions for your programs. After accelerating a program, you can strip the symbols region from it without affecting performance.

For a complete description of accelerating programs for TNS/R systems, see the *Accelerator Manual*. For a complete description of accelerating programs for TNS/E systems, see the *Object Code Accelerator*

(OCA) Manual. For a complete description of accelerating programs for TNS/X systems, see the *Object Code Accelerator for TNS/X (OCAX) Manual*. Topics covered in these manuals include:

- Determining which programs to accelerate
- Preparing your program for acceleration
- Specifying Accelerator or OCA program command-line syntax
- Setting Accelerator or OCA options
- Increasing the performance of accelerated programs
- Debugging accelerated programs

Working in the OSS Environment

From a G-series Open System Services (OSS) environment, you can compile, bind, and accelerate TNS C programs for either the OSS or Guardian environment. In the OSS environment, you use the TNS `c89` utility to invoke the C compiler, the Binder (the BIND program), and the Accelerator (the AXCEL program) to develop your applications. OCA is not available from the TNS `c89` utility.

This subsection is only a summary.

Versions of the c89 Utility

The TNS `c89` utility is located in the `/nonnative/bin` directory on D40 and later D-series and G-series software releases. By default, the native `c89` utility is run. You must set `/nonnative/bin` at the start of your `PATH` environment variable to get the TNS `c89` utility instead of the native `c89` utility.

For more details, see the TNS `c89` online reference page (`man -M /nonnative/usr/share/man c89`). The `c89` reference page in D40 and later versions of the *Open System Services Shell and Utilities Reference Manual* describes the native `c89` utility.

Components of the TNS c89 Utility

The TNS `c89` utility enables you to compile ISO-compliant C programs for NonStop environment. By default, `c89` generates code for the OSS environment. `c89` provides a simple interface to the components of the C compilation system:

- C language preprocessor

  The preprocessor manipulates the text of the source file to prepare it for the C compiler, including the insertion of function prototypes from the specified header files.

- C compiler

  The compiler translates the source text into object code.

- Binder

  The Binder resolves external references to library routines and performs fixups to the object file.

- Accelerator

  The Accelerator generates RISC instructions from the TNS object file.

- SQL/MP compiler

This compiler generates the appropriate code for embedded SQL/MP statements.

Using the TNS c89 Utility

Control TNS `c89` with flags and operands. Flags direct `c89` to initiate tasks, such as compiling a source file. Flags are also used to change the settings of components controlled by `c89`, such as to direct the C compiler to generate symbolic information used in debugging. Operands are the objects acted on or used by the components controlled by `c89`, such as source files to compile or library files to search when resolving external references. The simplified syntax of the TNS `c89` utility is:

```
c89 [flag...] operand...
```

flag

is a valid `c89` utility flag.

operand

is a valid `c89` utility operand.

Commonly Used TNS c89 Operands table summarizes commonly used operands.

Table 35: Commonly Used TNS c89 Operands

To specify this type of file	Specify this operand
A C source file	*filename.c*
An object file	*filename.o*

Commonly Used TNS c89 Flags and Guardian Environment Equivalents table summarizes commonly used TNS `c89` flags and describes equivalent Guardian environment actions.

Table 36: Commonly Used TNS c89 Flags and Guardian Environment Equivalents

To direct c89 to:	**Specify this flag:**	**Equivalent Guardian environment action**
Accelerate a program file	`-O`	Run the Accelerator program
Compile the specified source files, but do not bind them	`-c`	Run the C compiler without specifying the `RUNNABLE` pragma
Display detailed information from the C compiler, Binder, and Accelerator	`-Wverbose`	Default behavior
Pass an argument string to the Accelerator	`-Waxcel [="arguments"]`	Specify arguments on the Accelerator command line or in a command file

Table Continued

To direct c89 to:	Specify this flag:	Equivalent Guardian environment action
Pass an argument string to the Binder	`-Wbind [= "arguments"]`	Specify arguments in a Binder command file or run the Binder interactively
Pass an argument string to the C compiler	`-Wccom [= "arguments"]`	Specify pragmas on the C compiler command line or in a source file
Pass an argument string to the C language preprocessor	`-Wcprep [= "arguments"]`	No equivalent in the Guardian environment
Pass an argument string to the SQL compiler	`-Wsql [= "arguments"]`	Run the SQL compiler
Produce symbolic debugging information	`-g`	Specify the `SYMBOLS` pragma
Specify the module runs in the Guardian environment	`-Wsystype = "guardian"`	Specify the `SYSTYPE GUARDIAN` pragma on the C compiler RUN command line or by default
Specify the module runs in the OSS environment (default)	`-Wsystype = "oss"`	Specify the `SYSTYPE OSS` pragma on the C compiler RUN command line
Select dynamic binding	`-WBdynamic`	No equivalent in the Guardian environment
Select static binding	`-WBstatic`	Default behavior
Suppress the invocation of Binder	`-Wnobind`	Do not specify the `RUNNABLE` pragma
Use the pathname *outfile* instead of the default `a.out` for the executable file produced	`-o outfile`	No equivalent in the Guardian environment

Table Continued

To direct c89 to:	Specify this flag:	Equivalent Guardian environment action
Define a preprocessor symbol	`-D name`	Specify the `#define` preprocessor directive in the source file or use the C compiler command line define option
Undefine a preprocessor symbol	`-U name`	Specify the `#undef` preprocessor directive or use the C compiler command line undefine option
Change the search algorithm for header files	`-I directory`	Specify the `SSV` pragma
Change the search algorithm for library files	`-L` *directory*	Similar to specifying the `SEARCH` pragma

Binding an OSS TNS Module

By default, the TNS `c89` utility performs dynamic binding. In dynamic binding, the Binder resolves references to library functions using a TNS shared run-time library (SRL). Final resolution of references is performed at run time. A TNS shared run-time library (SRL) is a special form of a TNS user library that can contain global variables. Programs that use an SRL cannot use a user library. HPE provides TNS SRLs for OSS and NonStop TUXEDO. You cannot build your own TNS SRLs. Do not confuse TNS SRLs with the TNS/R native SRLs provided on D40 and later software releases. For information on native SRLs, see **Shared Run-Time Libraries (SRLs)** on page 377.

TNS `c89` uses the SRL `libc.so` in the OSS file system by default. `libc.so` is equivalent to the libraries `libc.a` and `libm.a` in a UNIX environment and COSS in the Guardian environment.

Dynamic binding produces smaller program files and uses fewer system resources than static binding.

You can also perform static binding for OSS programs. In static binding, the Binder resolves references to library functions by binding the functions into the program. Static libraries can be used for dynamic binding. The Binder resolves external references using all the specified static libraries before using the SRL.

Binding Considerations

- To bind a Guardian program with the TNS c89 utility, specify the libgwc.a library. libgwc.a is equivalent to the Guardian file CWIDE. The -Wsystype=guardian flag sets the default library to libgwc.a.
- You cannot use an SRL for Guardian programs. Only OSS programs can use an SRL.
- You must bind internationalized programs using an SRL that supports internationalization. For more details on binding internationalized programs, see the *Software Internationalization Manual*.

Examples of Working in the OSS Environment

These examples show you how to use the TNS c89 utility to compile, bind, and accelerate programs in the OSS environment.

1. In this example, c89 compiles source file test1.c and binds the object file into program file a.out. The default shared run-time library libc.so is used to resolve external references with dynamic binding:

   ```
   c89 test1.c
   ```

2. In this example, c89 compiles source file test2.c into object file test2.o:

   ```
   c89 -c test2.c
   ```

3. In this example, c89 compiles source file test3.c, binds the resultant object file, and accelerates the program file into an accelerated program file a.out. Binder uses the default shared run-time library libc.so to resolve external references, and the compiler generates a symbols region for symbolic debugging:

   ```
   c89 -g -O test3.c
   ```

4. In this example, c89 invokes the SQL compiler for the object file test4.o:

   ```
   c89 -Wsql test4.o
   ```

5. In this example, c89 compiles source file test5.c and binds the object file into program file testprog. Binder uses the library libmine.a to resolve external references before the library libc.a. Because libmine.a and libc.a are not shared run-time libraries, static binding is performed:

   ```
   c89 -o testprog -l mine -l c test5.c
   ```

6. In this example, c89 compiles the source file gprogram.c and binds the object file into program file a.out. Binder uses the default library for the Guardian environment, libgwc.a, to resolve external references. Static binding is performed:

   ```
   c89 -Wsystype=Guardian gprogram.c
   ```

7. In this example, c89 produces a statically bound program:

   ```
   c89 -o test3 -O -D TYPE=3 -I /usr/myself/header
      -I /usr/friend -WBstatic x1.c x2.o x3.c -l mylib
   ```

   The command compiles source files x1.c and x3.c and binds the object files together with x2.o into program file test3. During compilation, the preprocessor symbol TYPE is defined and is assigned the value 3. The compiler looks for header files in directory /usr/myself/header first, then it looks in /usr/friend, and finally it looks in /usr/include. Static binding has been specified, so the

Binder tries to resolve references using the library `mylib.a` before using the standard library `libc.a`. The `-O` flag causes invocation of the Accelerator on the program file.

8. In this example, `c89` produces a program made up of modules compiled for both the Guardian and OSS environments:

   ```
   c89 -Wsystype=guardian -o guard.o -c guard1.c guard2.c
   ```

   The command compiles the source files `guard1.c` and `guard2.c` for the Guardian environment and generates an object file `guard.o`. The `-c` flag suppresses the binding phase. The command:

   ```
   c89 oss1.c oss2.c guard.o
   ```

   compiles the source files `oss1.c` and `oss2.c` for the OSS environment, binds the resulting object files with the object file `guard.o` to generate the program file `a.out`.

Compiling, Binding, and Accelerating TNS C++ Programs

Working in the Guardian Environment

To create an executable TNS C++ program in the Guardian environment:

1. Run the C preprocessor Cprep
2. Run the C++ translator Cfront
3. Run the C compiler
4. Run the Binder
5. Run the Accelerator or Object Code Accelerator (OCA) (optional)

This chapter describes each compilation step, including the purpose of the compilation, the output from the compilation, and the syntax description for the compilation run command. This section also contains an example that shows you how to compile a simple TNS C++ program.

You can use a separate run command for each of the C++ compilation steps, as described in this section and shown in the compilation example, or you can use a TACL macro to consolidate these compilation steps.

If you use a TACL macro to drive the C++ compilation, you still need to understand what is needed for each compilation step, what is accomplished by each compilation step, and what is the possible output from each compilation step. This section provides you with this information.

After compiling your C++ program, you run the Binder to create an executable C++ object file. This section describes a sample set of Binder commands that you can use to bind your C++ programs.

You can use the Accelerator or OCA to make programs more efficient on native systems. The same rules that apply to accelerating C programs apply to accelerating C++ programs. If you want to accelerate your program, see the *Accelerator Manual or Object Code Accelerator (OCA) Manual* for the necessary information.

Cfront is shipped with a simple TACL macro, named cplus, that allows you to consolidate the compilation process. The cplus macro is also provided here for your information:

```
?tacl macro

==This is a general purpose macro to use with Cfront on an HP™
System. You can add any preprocessor options such as
== SSV pragmas to this macro.
==
purge %1%1, %1%2, %1%o, %1%L
#output Preprocessing %1%c
cprep /in %1%c, out %1%1/ define __cplusplus, ssv0 "$system.system"
[#if NOT(:_COMPLETION:COMPLETIONCODE>1) |THEN|
   #output Finished Preprocessor step.  Starting CFRONT.
   cfront /in %1%1,  out %1%2/
   [#if NOT(:_COMPLETION:COMPLETIONCODE>1) |THEN|
      #output Finished CFRONT step.  Starting C.
      c /in %1%2, out %1%L/%1%o; suppress
      [#if NOT(:_COMPLETION:COMPLETIONCODE>1) |THEN|
         #output Finished C step. Purging intermediate files.
         purge %1%1, %1%2, %1%L
      ]
   ]
]
```

If the C++ files Cprep, Cfront, and the C compiler are installed on $SYSTEM, the TACL macro cplus should be in $SYSTEM.ZCFRONT.

To use this macro, the name of your C++ source file must end in the letter “c.” For example, the command syntax to compile a C++ program located in file `helloc` is:

```
cplus hello
```

If the name of your C++ source file does not end in the letter “c,” you should either rename your source file or modify the cplus macro.

C Preprocessor Cprep

Cprep is a general-purpose C preprocessor. Cprep performs these functions:

- Expands all preprocessor directives
- Uses certain C compiler pragmas and passes the rest to Cfront for the C compiler
- Generates `#line` directives specifying original C++ source file and edit line number
- Produces a TNS C++ source file that is the input for Cfront

Expansion of Preprocessor Directives

Cprep includes files as directed by `#include` directives and `SSV` compiler pragmas. If any `SSV` compiler pragmas are specified, they must appear on the Cprep run command line. In addition, Cprep expands all the preprocessor directives.

C Compiler Pragmas Used by Cprep

Cprep recognizes certain C compiler pragmas. These pragmas are used by Cprep and where relevant they are passed through. These pragmas must appear on the Cprep run command line or in the C++ source code. Cprep recognizes these pragmas. **C Compiler Pragmas Used by Cprep** table lists the C compiler pragmas used by Cprep.

Table 37: C Compiler Pragmas Used by Cprep

[NO]CHECK	RUNNABLE
COLUMNS	SECTION (source code only)
ERRORS	SSV (command line only)
[NO]NEST	[NO]WARN
[NO]OLDCALLS	[NO]WIDE

All other C compiler pragmas are passed through to the file created by Cprep and are used by the C compiler.

The TNS C++ translator, Cfront, is WIDE by default. The TNS C compiler is effectively NOWIDE by default. To compile TNS C++ programs that use the 32-bit data model for the Guardian environment, specify pragma WIDE in the C++ source file or on the TNS C++ preprocessor, Cprep, run command line.

For a complete description of all the C compiler pragmas, see **Compiler Pragmas** on page 193.

Generation of #line Directives

Cprep generates #line directives indicating the original C++ source file, edit line number, and timestamp. These #line directives appear in the output from Cprep and are also propagated through to the output of Cfront. These #line directives are generated to support source-level debugging and to support generation of error messages from Cfront. The symbolic debuggers use the #line directive information to map from object code to the corresponding location in the C++ source code.

The first time a file is referenced, the #line directive has the form:

```
#line edit-line-number "source-file-name timestamp"
```

For subsequent lines in that file, the #line directive has the form:

```
#line edit-line-number
```

The timestamp given the first time a file is referenced represents the modification timestamp for the C++ source input file. This information is used by the symbolic debuggers to identify possible source code version inconsistencies.

Cprep Run Command Syntax

To notify Cprep that it is processing a C++ program, you must specify #define __cplusplus in the source file or define __cplusplus on the Cprep run command line. Note that two underscore characters are required before the word cplusplus.

Cprep creates an intermediate file to be used by Cfront. Cfront creates an intermediate file to be used by the C compiler. Be sure to purge any existing intermediate files before running Cprep. If you forget to delete an existing intermediate file, the output will be appended to the end of that file.

You use this syntax to run Cprep:

```
CPREP /IN source, OUT intermediate-file1 / define __cplusplus
     [ , compile-option [ , compile-option ]...]

  compile-option:

     { SSV-pragma                         }
     { pragma                             }
     { define identifier [ integer-constant ] }
```

IN *source*

specifies the name of the file that contains your original C++ source code.

OUT *intermediate-file1*

specifies the file to which Cprep writes its C++ source file. If you omit the OUT option, the output goes to your current default output file, usually the terminal.

define __cplusplus

specifies to Cprep that it is processing a C++ program.

compile-option

modifies compiler operation by specifying a compiler pragma or defining a preprocessor symbol.

SSV-pragma

is a C compiler pragma that specifies a subvolume in which a header file is located. For more details, see **SSV** on page 308.

pragma

is a C compiler pragma.

These TNS C compiler pragmas are intended for Cprep and can appear only in this Cprep run command syntax or in the TNS C++ source code:

`[NO]CHECK`	`RUNNABLE`
`COLUMNS`	`SECTION` (source code only)
`ERRORS`	`SSV` (command line only)
`[NO]NEST`	`[NO]WARN`
`[NO]OLDCALLS`	`[NO]WIDE`

Pragmas `NOXMEM` and `SQL` are invalid for C++. If either of these pragmas appears, Cfront issues a warning and ignores the pragma.

For a detailed description of all the C compiler pragmas, see **Compiler Pragmas** on page 193.

define *identifier [integer-constant]*

defines *identifier* as a preprocessor symbol. If `identifier` is followed by an integer or character constant, `identifier` is defined as an object-like macro that expands to the given value.

Usage Guidelines

- Remember that the output that is produced from running Cprep is kept in an intermediate file. Each time that you run Cprep, the output is added to the end of the file. Therefore, if intermediate files exist, purge them before running Cprep again.
- The pragmas that you specify for Cprep cannot be abbreviated. You must type out the full name for the pragma.
- Cprep and Cfront both send error and warning messages to `stderr` . The default location for `stderr` is the terminal. You can assign `stderr` to a specific file and collect all Cprep and Cfront error and warning messages there.

  To send error and warning messages to a file, assign `stderr` to the desired file name prior to invoking Cprep. You assign `stderr` to a file name:

  ```
  ASSIGN STDERR, file-name
  ```

  To return to the default location, enter:

  ```
  CLEAR ASSIGN STDERR
  ```

Example

In this example, the C++ source file progcp is input to Cprep. An intermediate file, intfile1, is created by Cprep for later use by Cfront. The SSV pragma SSV0 causes any header files in progcp to be searched for in the subvolume hdrvol; that is, Cprep looks in subvolume hdrvol for a file named hdrh if progcp contains the statement #include "hdrh."

```
CPREP/IN progcp, OUT intfile1/define __cplusplus, SSV0 "hdrvol"
```

C++ Translator Cfront

Cfront generates code for the TNS C compiler. In so doing, Cfront performs these functions:

- Always generates function prototypes
- Uses the NOWIDEor RUNNABLE pragmas, if they were specified on the Cprep run command line or in the C++ source
- Generates these pragmas: C_PLUS_PLUS, C_PLUS_PLUS_STMT, NOWARN(134, 135), and SEARCH
- Produces a C source file that serves as the input to the TNS C compiler

Pragmas Generated by Cfront

Cfront generates these pragmas: C_PLUS_PLUS, C_PLUS_PLUS_STMT, NOWARN(134, 135), and SEARCH. Each of these pragmas is explained below.

Pragma	Use
C_PLUS_PLUS	notifies the TNS C compiler that this object file is derived from C++ source code.
C_PLUS_PLUS_STMT	identifies a C++ statement to the TNS C compiler. This pragma causes the next C statement to be identified as the start of generated C code for the C++ statement to be found in the file and at the line number specified by the latest #line directive. This information is used for symbolic debugging.

Table Continued

Pragma	Use
NOWARN(134, 135)	disables the TNS C compiler from generating warning messages 134 and 135. C compiler messages 134 and 135 warn that a function was used before it was declared. Cfront turns off these warning messages because the #include preprocessor directives in your C++ source code that declare the standard C header files are expanded by Cprep. Therefore, the C compiler would not see the #include directives and would emit warning messages anytime your program calls a standard C function.
SEARCH	directs the TNS C compiler to search a given object file when attempting to resolve external references. If your program has multiple compilation units, use Binder to combine the C object files produced by the TNS C compiler into one executable C++ object file. If your program comprises a single compilation unit, you can use the RUNNABLE pragma in the C++ source file or on the Cprep run command line to direct the C compiler to produce an executable C++ object file without having to run the Binder.When you specify the RUNNABLE pragma, Cfront automatically generates the SEARCH pragma with either the C++ run-time library file libca or libla specified. Because Cfront generates the SEARCH pragma, you do not have to run Binder and specify the Binder SEARCH command for these files.

Cfront Run Command Syntax

You use this syntax to run Cfront:

```
CFRONT / IN intermediate-file1, OUT intermediate-file2 /
```

IN *intermediate-file1*

specifies the name of the file that contains the C++ source code that was output by Cprep.

OUT *intermediate-file2*

specifies the file to which Cfront writes the C source file. If you omit the OUT option, the output goes to your current default output file, usually the terminal.

Usage Guidelines

- Remember that the output that is produced from running Cfront is kept in an intermediate file. Each time that you run Cfront, the output is added to the end of the file. Therefore, if intermediate files exist, purge them before running Cfront again. The easiest way to do this is to purge all intermediate files before running Cprep.
- Cprep and Cfront both send error and warning messages to `stderr`. The default location for `stderr` is the terminal. You can assign `stderr` to a specific file and collect all Cprep and Cfront error and warning messages there. To send error and warning messages to a file, assign `stderr` to the desired file name prior to invoking Cprep. You assign `stderr` to a file name:

  ```
  ASSIGN STDERR, file-name
  ```

  To return to the default location, enter:

  ```
  CLEAR ASSIGN STDERR
  ```

Example

In this example, the intermediate file `intfile1`, which was previously created by Cprep, is now input to Cfront. An intermediate file, `intfile2`, is created by Cfront for later use by the C compiler.

```
CFRONT/ IN intfile1, OUT intfile2 /
```

TNS C Compiler Run Command Syntax

The TNS C compiler compiles the output from Cfront and produces a compiler listing and a TNS object file, provided that the compiler encounters no errors during the compilation.

If your program has multiple compilation units, you use Binder to combine the TNS object files produced by the C compiler into one executable TNS object file.

If your program comprises a single compilation unit, you can use the `RUNNABLE` pragma in the C++ source file or on the Cprep run command line to direct the TNS C compiler to produce an executable TNS object file without having to run the Binder.

You use this syntax to run the TNS C compiler:

```
[ RUN ] C / IN intermediate-file2, OUT listing ]

   [ , run-options ] / [ object ] [ ; pragma [ , pragma ]...]
```

IN *intermediate-file2*

specifies the name of the file that contains the output from Cfront.

OUT *listing*

specifies the file to which the TNS C compiler writes the compiler listing. When specified, *listing* is usually a spooler location. If you omit the OUT option, the compiler writes the listing to your current default output file, usually the terminal.

When you run the TNS C compiler, error and warning messages are sent to *listing* (`stdout`), not `stderr`.

run-options

is a comma-separated list of additional options for the RUN command. These options are described in the TACL Reference Manual.

Object

specifies the file to which the TNS C compiler writes the TNS object code. If you do not specify an object file, the TNS C compiler writes the TNS object code to the file OBJECT in your current default volume and subvolume. If OBJECT cannot be created, the compiler writes the object code to the file ZZBI*nnnn* (where *nnnn* is a unique four-digit number) in your current default volume and subvolume.

pragma

is a C compiler pragma.

These TNS C compiler pragmas are intended for Cprep and can appear only in this Cprep run command syntax or in the C++ source code:

`[NO]CHECK`	`RUNNABLE`
`COLUMNS`	`SECTION` (source code only)
`ERRORS`	`SSV` (command line only)
`[NO]NEST`	`[NO]WARN`
`[NO]OLDCALLS`	`[NO]WIDE`

Pragmas `NOXMEM` and `SQL` are invalid for C++. If either of these pragmas appear, Cfront issues a warning and ignores the pragma.

Usage Guideline

You can place pragmas intended for the TNS C compiler on the TNS C compiler run command line. However, this practice is a likely source of errors in consistency and configuration management, because these same pragmas can also appear on the Cprep run command line. Therefore, whenever possible, you should place pragmas in the original C++ source code or on the Cprep run command line. Cprep passes the pragmas intended for the TNS C compiler through to Cfront for the TNS C compiler.

Example

In this example, the intermediate file intfile2, which was previously created by Cfront, is now input to the TNS C compiler. The TNS C compiler produces the TNS object file `progo` that you can now use to bind together with other TNS object files to produce an executable TNS object file for the C++ program.

```
C/ IN intfile2, OUT $s.#hold/progo
```

File Formats

The input to Cprep is a C++ source file of a form acceptable to the TNS C run-time library:

- EDIT disk files, which are file type 101
- C disk files, which are odd-unstructured files and are file type 180

- Processes
- Terminals

The input to Cprep is usually an EDIT file. The output of Cprep is a C++ source file, which is referred to in this section as an intermediate file. This intermediate C++ source file is by default a type 101 file and is used as the input to Cfront.

The output of Cfront is a C source file, which is also referred to in this section as an intermediate file. This intermediate C source file is by default a type 180 file and is used as the input to the TNS C compiler.

If you want your intermediate file to be a file type other than the default file type, create an empty file of the desired type before running Cprep or Cfront. The file you create must be acceptable to the TNS C run-time library.

If an intermediate file exists prior to the invocation of Cprep or Cfront, the output is appended to the end of that file.

If you do not specify an output file to Cprep or Cfront, the output will go to the terminal.

Compiling a Sample C++ Program

This example shows a C++ source program and the run commands necessary to compile it. The C++ source program comprises a single compilation module, so you can compile it with the `RUNNABLE` pragma specified. The compilation will produce an executable C++ TNS object file, so you do not have to run the Binder. Here is the C++ source program:

```
#pragma runnable
#include "iostream.h"   // Cprep truncates to iostreah

main()

   cout << "Hello World\n";
}
```

This example assumes:

- The C++ source program is located in a file named progcp
- The file named `intfile1` is the intermediate file that will contain the C++ source code created by Cprep
- The file named `intfile2` is the intermediate file that will contain the C source code created by Cfront

Before compiling this program, purge any existing intermediate files:

```
purge intfile1
purge intfile2
```

To compile this program, enter these run commands:

```
cprep/in progcp, out intfile1/define __cplusplus, &
   SSV0 "$system.system"
cfront/in intfile1, out intfile2/
c/in intfile2, out $s.#hold/progo
```

To run this program, enter:

```
run progo
```

The program will respond with:

```
Hello World
```

Error Messages in the Guardian Environment

This subsection provides information on the compile-time and run-time error messages that are generated in the Guardian environment.

Compile-Time Error Messages

Cprep and Cfront generate completion codes and send error and warning messages to `stderr`.

Cprep and Cfront each follow the standard HPE convention of generating completion codes to indicate the presence of error or warning messages. The completion codes have these significance:

Completion Code	Significance
1	Compilation completed with warnings, but no errors.
2	Errors occurred, and processing terminated.

If you are running Cprep or Cfront interactively, these completion codes appear on your terminal screen as a source of information. For example, if Cfront encounters errors and stops processing, it will emit this message:

```
2: Process terminated with fatal errors or diagnostics
```

If you are running in batch mode, Cprep and Cfront sends completion codes to the command interpreter. You can use TACL to check for these completion codes. For an example of how to check for these completion codes, see the TACL macro in **TNS C++ Implementation-Defined Behavior** on page 543.

Cprep and Cfront both send error and warning messages to `stderr`. The default location for `stderr` is the terminal. You can assign `stderr` to a specific file and collect all Cprep and Cfront error and warning messages there. The messages generated by Cfront contain (CFRONT): at the start of the message.

The total count of errors and warnings appears as a comment after the error and warning messages.

Cprep's error and warning messages that appear in `stderr` are a subset of those of the TNS C compiler. For descriptions of the TNS C compiler error and warning messages, see **TNS C Compiler Messages** on page 438.

Once Cfront recognizes errors, it stops emitting source code, and after a maximum of 13 errors, it stops compiling. When Cfront stops compiling, it terminates with the message `Process terminated with fatal errors or diagnostics`. Cfront's error and warning messages that appear in `stderr` are self-explanatory.

Run-Time Error Messages

C++ uses the TNS C run-time library. For a description of the TNS C run-time error messages, see **TNS C Compiler Messages** on page 438.

Binding C++ Programs

If you do not use the `RUNNABLE` pragma when compiling a single-module program or if your program comprises several separately compiled modules, you must use Binder to collect and combine the compiled modules into an executable C++ object file.

To start Binder, enter the BIND command at the TACL prompt:

```
10> BIND
```

Binder Commands

After starting Binder, you enter Binder commands to combine your compiled modules with the TNS C++ run-time library object files and with the TNS C run-time library object files to produce an executable C++ TNS object file. The Binder commands you can use are shown in this diagram.

```
SELECT RUNNABLE OBJECT ON
SELECT CHECK PARAMETER OFF
SET INSPECT ON
ADD * FROM main-object-file
ADD * FROM object-file
SELECT SEARCH C++-library-file
SELECT SEARCH data-model-file
SET HEAP value PAGES
BUILD program-file
```

SELECT RUNNABLE OBJECT ON

directs Binder to create an executable object file (a program file) when it builds the object file.

SELECT CHECK PARAMETER OFF

disables checking of parameter number, type, and mode (value or reference) and of function return type across code blocks. If you do not disable these checks, Binder might generate several extraneous warning messages.

SET INSPECT ON

specifies that the currently defined symbolic debugging program, rather than the default debugging program, is chosen for debugging when you run the object file.

ADD * FROM *main-object-file*

adds the object file of your program's main module. This main module is the one that contains the definition of the function `main()`.

ADD * FROM *object-file*

adds the object file of one of your program's other modules. You must repeat this command for each of your modules.

SELECT SEARCH *C++-library-file*

directs Binder to search the appropriate C++ run-time library file when resolving external references.

Specify LIBCA if your program uses the 32-bit data model. For the 32-bit data model, the size of the type int is 32 bits.

Specify LIBLA if your program uses the 16-bit data model. For the 16-bit data model, the size of the type int is 16 bits.

SELECT SEARCH *data-model-file*

directs Binder to search the specified data-model file when resolving external references.

Use CWIDE for the *data-model-file* to specify the 32-bit data model.

Use CLARGE for the *data-model-file* to specify the 16-bit data model.

The *data-model-file* file name must be fully qualified. CWIDE and CLARGE are in $SYSTEM.SYSTEM by default.

SET HEAP *value PAGES*

specifies the heap size of the program.

BUILD *program-file*

directs Binder to build an executable object file using the files and options specified in the preceding commands. The *program-file* is the name of the executable object file that Binder builds.

For more details regarding Binder and its commands, see the *Binder Manual*.

Requirements for Binding Modules

These requirements apply when you are binding two or more compiled modules to create a single executable C++ object file:

- Each of the C++ modules must be based on the same data model. If any module has been compiled with the `NOWIDE` pragma specified, all modules must be compiled with the `NOWIDE` pragma specified. Likewise, if any modules have been compiled with the `WIDE` pragma specified, all modules must be compiled with the `WIDE` pragma specified.
- If you bind C++ modules with C modules, each of the modules must be based on the same memory model and the same data model.

Working in the G-Series OSS Environment

The TNS `c89` utility enables you to compile TNS C++ programs in the Open System Services (OSS) environment. The `c89` utility is XPG4 compliant. This section gives an overview of the standard file operands and flags, lists the HPE extensions that are applicable to TNS C++, and gives some examples.

The TNS `c89` utility is located in the `/nonnative/bin` directory on D40 and later software releases. By default, the native `c89` utility is run. You must set `/nonnative/bin` at the start of your `PATH` environment variable to get the TNS `c89` utility instead of the native `c89` utility.

For more details, see the TNS `c89` online reference page. To view this reference page, enter:

```
man -M /nonnative/usr/share/man c89
```

The `c89(1)` reference page in D40 and later versions of the *Open System Services Shell and Utilities Reference Manual* describes the native `c89` utility.

The TNS `c89` utility provides a simple interface to the components of the C++ compilation system. This compilation system conceptually consists of:

- Cprep
- Cfront
- C compiler
- Binder
- Accelerator, an optional program component

You control the actions performed by TNS `c89` with file operands and flags. When the file operand is a C++ source file, TNS `c89` invokes Cprep, Cfront, and the C compiler to process the file. The preprocessor symbol `__cplusplus` is automatically defined in this case. A file operand is perceived by TNS `c89` as being a C++ source file when the `-Wcfront` flag is specified or when the file-name extension is `.C` or `.cpp`.

The simplified syntax of the TNS `c89` utility is:

```
c89 [flag...] operand...
```

flag

is a valid `c89` utility flag.

operand,

is a valid `c89` utility operand.

TNS c89 Flags

You can invoke Cfront through the TNS `c89` utility with the -Wcfront flag using this syntax:

```
-Wcfront[="args"]
```

When TNS `c89` sees the ="*args*" portion of the -Wcfront flag, `c89` passes to Cprep the argument string enclosed in the double quotation marks. **Commonly Used TNS c89 Flags** table lists other commonly used TNS `c89` flags.

Table 38: Commonly Used TNS c89 Flags

Flag	Function
`-c`	Compiles the specified source files but does not bind them.
`-E`	Runs only Cprep and directs output to the standard output file.
`-o outfile`	Specifies the pathname *outfile* , instead of the default `a.out.`
`-O`	Generates an accelerated object file.
`-Wcfonly`	Stops processing the source file after Cfront is done, and sends output from Cfront to the standard output file.
`-Wcfront="`*`arguments`*`"`	Passes argument strings to Cprep and Cfront. If the -Wcfonly flag is also specified, output from Cfront is sent to the standard output file and no compilation is performed. Otherwise, output from Cfront is sent to the C compiler.
`-Wcprep="`*`arguments`*`"`	Passes an argument string to Cprep, and stops processing the source file after this preprocessing stage.
`-Wsystype=guardian`	Specifies the Guardian execution environment.

Table Continued

Flag	Function
`-Wsystype=oss`	Specifies the OSS execution environment (this is the default environment).
`-Wverbose`	Displays more detailed information during the program generation process from the C compiler, Binder, Accelerator, and SQL compiler.In particular, Cfront does not default to sending the Cfront banner or error counts to `stderr` when running in the OSS environment. Specify the -Wverbose flag to cause the banner and error counts to be sent to `stderr` .

TNS c89 Operands

An operand is in the form of either a pathname or `-l` *library.* Note also that TNS `c89` recognizes the operands in **C and C++ Extensions** on page 53.

Table 39: TNS c89 Operands

Operand	Function
file.C	A C++ language source file to be processed by Cprep and Cfront, compiled by the C compiler, and optionally processed by the Binder.
file.cpp	A C++ language source file to be processed by Cprep and Cfront, compiled by the C compiler, and optionally processed by the Binder. (This operand is the same as *file* `.C`.)
file.a	A library of object files typically produced by the `ar` utility and passed directly to the Binder.
file.o	An object file produced by `c89 -c` and passed directly to the Binder.
file.so	A shared run-time library produced by Binder. The shared run-time library is used by the Binder to resolve external references.
`-l` *library*	In the static binding mode, search for the library named `lib`*library.a*. In the dynamic binding mode, search for the library named `lib`*library.so*. If `lib`l *ibrary.so* is not found, `lib`*library.a* is used. For example, if you use the operand -l cre,the libraries searched for are libcre.a and libcre.so.A library is searched when its name is encountered, so the placement of a `-l` operand is significant.

Input Files

Input files to TNS `c89` can be:

- C++ language source files
- TNS object files generated by c89 -c
- Libraries of TNS object files produced by the ar utility
- Libraries of TNS object files produced by Binder
- Executable TNS object files produced by Binder

Binding

The -lc and -lC operands specify these libraries:

-lc	This library contains all library functions specified in the POSIX.1 specification, except for those functions listed as residing in math.h.
-lC	This is the TNS C++ run-time library. It should be bound before the standard TNS C run-time library.

In the absence of options that inhibit the invocation of the Binder, such as -c or -Wnobind, the TNS c89 utility causes the equivalent of a -lc operand to be passed to the Binder as the last -l operand, causing the standard TNS C library to be searched after all other object files and libraries are loaded.

If one or more of the file operands are C++ source files, TNS c89 causes the equivalent of a -lC operand to be passed to the Binder ahead of -lc. However, if a separate binding process is invoked to bind C++ TNS object files, you are required to specify -lC as one of the file operands ahead of -lc. If you do not specify -lC ahead of -lc only the standard TNS C run-time library is used by the Binder.

Examples

Examples 1 and 2 shows how to compile a C++ program using the TNS c89 utility. Example 3 shows how to perform some of the compilation steps without performing all of them.

1. This example preprocesses hello.C with Cprep, translates it with Cfront, compiles it with TNS C, and then binds it with the Binder to create an executable file named a.out.

   ```
   c89 hello.C
   ```

2. This example causes the same compilation steps to occur as in the previous example except that the resulting executable file is named hello.

   ```
   c89 -o hello hello.C
   ```

3. This example preprocesses hello.C with Cprep and translates it with Cfront. The output from Cfront is saved in a file named hello.i. No compilation by the TNS C compiler is performed because the -Wcfonly flag has been specified.

   ```
   c89 -Wcfonly -o hello.i hello.C
   ```

Error Messages in the OSS Environment

When the TNS c89 utility encounters a compilation error that causes an object file to not be created, it writes a diagnostic to the standard error file and continues to compile other source code operands. TNS c89 does not perform program binding and returns a nonzero exit status.

When a Binder operation is unsuccessful, a diagnostic message is written to the standard error file and TNS c89 exits with a nonzero status.

When an Accelerator operation fails, TNS `c89` exits with a nonzero status and the content of the program file is unchanged.

Compiling and Linking TNS/R Native C and C++ Programs

The TNS/R native C and C++ compilers take as input a module (a translation unit) and generate an object file or linkfile. A module is defined as a source file with all the headers and source files it includes, except for any source lines skipped as the result of conditional preprocessor directives.

The `nld` utility links object files generated by the compilers and produces an executable object file (a program file).

The `ld` utility is the TNS/R native linker that links PIC (Position-Independent Code) linkfiles to produce PIC loadfiles.

The SQL compiler processes executable files and generates code for embedded SQL statements.

The native `c89` and `c99` utilities control the C and C++ compilation system in the Open System Services (OSS) environment. Note that the `c89` utility on TNS/E systems can generate TNS/R code for linking by the `ld` utility (not the `eld` utility) when the `-Wtarget=tns/r` flag is used; discussions of OSS `c89` in this section therefore apply to TNS/E users when that flag is specified.

For information on how to compile and link programs in the OSS environment, see the `c89(1)` reference page online or in the *Open System Services Shell and Utilities Reference Manual*. For more detail on converting from TNS `c89` to TNS/R native `c89`, see the *TNS/R Native Application Migration Guide*.

For information about compiling and binding native C++ programs using the Windows PC environment, see the online help for the Enterprise Tool Kit. In this manual, the Enterprise Tool Kit is introduced in **Using the Native C/C++ Cross Compiler on the PC** on page 419.

Selecting a Development Platform

A development platform consists of the hardware system and software environment available to compile, link, and run a program.

You can develop OSS programs regardless of whether the OSS environment is available on the system. However, you cannot run and test OSS programs on a system without the OSS environment.

It is easier to develop a program in the environment in which it runs, but you can develop a program in one environment that runs in another environment, with a few restrictions. However, compile time is lesser in the PC environment.

Development Platform Capabilities (TNS/R Native C and C++ Programs) table describes the capabilities of each development platform.

Table 40: Development Platform Capabilities (TNS/R Native C and C++ Programs)

Capability	System with Guardian environment	System with Guardian and OSS environments	ETK on Windows PC	NSDEE on Windows PC
Use Guardian development tools for Guardian programs?	Yes	Yes	No	Yes

Table Continued

Use Guardian development tools for OSS programs?	Yes	Yes	No	Yes
Use OSS development tools for Guardian programs?	No	Yes	No	Yes
Use OSS development tools for OSS programs?	No	Yes	No	Yes
Run Guardian programs?	Yes	Yes	No	Yes
Run OSS programs?	No	Yes	No	Yes
Compile programs with embedded SQL?	Yes	Yes	Yes	Yes
Use PC-based development tools for Guardian and OSS programs?	No	No	Yes	Yes

These restrictions apply to developing Guardian programs with OSS tools:

- You cannot use the `RUNNABLE` and `SEARCH` pragmas. However, you can direct the `c89` utility to bind implicitly after a compilation. You can also specify library files to be searched using the `c89-L` flag.
- You cannot use the `SSV` pragma. However, you can specify search directories using the `c89-I` flag.

Specifying Header Files

The macros and functions in the C run-time library are declared in header files. Each header file contains declarations for a related set of library functions and macros, in addition to variables and types that complete that set. If you use a function in the library, you should include the header file in which it is declared. You should not declare the routine yourself because the declarations in the header files have provisions for several situations that can affect the form of a given declaration, including:

- Whether the routine is implemented internally as a function or a macro
- Whether the function is written in a language other than C
- Whether you are compiling for the small- or large-memory model
- Whether you are compiling for the 16-bit or 32-bit (wide) data model
- Whether you are compiling for the NonStop environment
- Whether you are compiling for TNS mode or native mode

In addition, the header file prototype declarations enable the compiler to check parameters and arguments for compatibility, ensuring that function calls provide the correct number and type of arguments.

A single set of C library header files supports all three environments (Guardian, OSS, and PC) and both modes (TNS and native). The locations of the header files in the three environments is summarized in the **Locations of Header Files** table.

Table 41: Locations of Header Files

Environment	Location
Guardian	$SYSTEM.SYSTEM and $SYSTEM.ZTCPIP
OSS	`/usr/include` and its subdirectories
PC	ETK < version 3.0: `\Compaq ETK-NSE\`*`rel`*`\include` ETK Version 3.0 and later: `\Compaq ETK-NSE\`*`rel`*`\usr\include` where *rel* represents the release version update, such as D45.01

To specify header files, use the `#include` preprocessor directive. For example, to include the STDIO header file, specify in your object file:

```
#include <stdio.h>
```

You might also need to specify header files for the Standard C++ Library and Tools.h++ library to use functions contained in those libraries. For examples of `#include` directives for these libraries, see **Using the Standard C++ Library** on page 88 and **Accessing Middleware Using HPE C and C++ for NonStop Systems** on page 104. For information about including SRLs at link time, see **Determining Which SRLs are Required** on page 378.

You can specify locations to search for header files:

- In the Guardian environment, use the `SSV` pragma to specify a search list of subvolumes for files specified in `#include` directives. For more details, see pragma **SSV** on page 308.
- In the OSS environment, use the `-I` flag to the `c89` utility to specify a search list of directories for files specified in `#include` directives. For more details, see the `c89(1)` reference page either online or in the *Open System Services Shell and Utilities Reference Manual*.
- In the PC (Enterprise Tool Kit) environment, specify a search list of directories using the Directories page.
- In the PC command line environment (using the cross compilers), use the `-I` flag to the `c89` utility to specify a search list of directories for files specified in # `include` directives. For more details, see the document *Using the Command-Line Cross Compilers on Windows*.

While header files are optional (but strongly recommended) for programs that contain Guardian or OSS modules exclusively, header files are required for mixed-module programs. If you do not compile using header files, TNS/R native linker cannot correctly resolve external references to Guardian and OSS versions of C functions.

Compiling and Linking Floating-Point Programs

You can now choose either Tandem floating-point format or IEEE floating-point format for performing floating-point arithmetic in your native C and C++ programs. This table compares the two formats.

Tandem Floating-Point Format	IEEE Floating-Point Format
Is default for the TNS/R native compilers.	Is default for the TNS/E and TNS/X native compilers.
Is a proprietary implementation of floating-point arithmetic that is supported in software millicode	Is an industry-standard data format that is supported in the processor hardware
Provides backward compatibility with pre-G06.06 C and C++ applications	Requires the G06.06 release of the HPE NonStop OS and the G06.06 or later product version of the native C and C++ compilers
Remains default format on TNS and TNS/R systems, and is available for TNS C and C++, FORTRAN, TAL, pTAL, Pascal, COBOL, and native C and C++ programs	For TNS/R native C and C++ programs, IEEE floating-point format becomes effective only by specifying `IEEE_FLOAT` in the command directive
Requires conversion routines for data interchange between Tandem format and IEEE format (see the *Guardian Procedure Calls Reference Manual* for more detail about conversion routines)	Provides easier data interchange with other systems using 64-bit and 32-bit IEEE floating-point data formats; interchange typically consists of reversing the byte order to convert between big-endian data format (on NonStop systems) and little-endian format (on the target system)
	Provides better handling of exception conditions such as overflow and underflow; the existence of NaN (not a number), infinities, and exception flags make it easier to detect invalid results

Using Compiler Pragmas IEEE_Float and Tandem_Float

- To use IEEE floating-point format, you must specify the IEEE_FLOAT pragma on the command line when running the native C or C++ compiler. If you are using native C++, you also need to specify the `VERSION2` or `VERSION3` directive. For more details, see the pragmas **IEEE_FLOAT** on page 248, **VERSION2** on page 324, and **VERSION3** on page 326.
- To use Tandem floating-point format, you can optionally specify the TANDEM_FLOAT pragma (TANDEM_FLOAT is the default setting) on the command line when running the native C or C++ compiler. See the pragma **TANDEM_FLOAT** on page 319.

The native compilers set the floating-point format type in the generated object file.

Two examples of compiling native C and C++ programs that use IEEE floating-point format:

```
  NMC / IN SOURCEA, OUT $.#LIST / OBJECTA; IEEE_FLOAT

> NMCPLUS / IN SOURCEB, OUT $S.#LIST / OBJECTB; VERSION2, &
  IEEE_FLOAT
```

Using Link Options to Specify Floating-Point Format

When linking object files using the TNS/R native linker utility, you can specify the floating-point state of the output object file using the `-set floattype` flag. You can set any of these three options: `TANDEM_FLOAT`, `IEEE_FLOAT`, or `NEUTRAL_FLOAT`. If the -set `floattype` flag is not specified, the

TNS/R native linker utility derives the floating-point state of the output object file from the states of the input files.

When modifying an existing object file, TNS/R native linker sets the state as specified by the −change floattype flag.

Link-Time Consistency Checking

The TNS/R native linker utility checks the consistency of floating-point type combinations when linking object files. The checking differs according to whether the -set flag is specified.

When the floattype attribute is not explicitly set with the -set flag, TNS/R native linker uses the floattype attribute values of all the input object files for determining the floattype attribute value for the output object file. If the consistency checks of the input object files result in an invalid floating-point state or inconsistent value, an error message is generated and no output object file is created.

Floating Point Consistency Check by TNS/R Native Linker Utility shows the results of each combination of floating-point states when the floattype attribute is not explicitly specified. In the columns labeled IEEE, Tandem, and Neutral are the number of input object files characterized by that floating-point setting.

Table 42: Floating Point Consistency Check by TNS/R Native Linker Utility

Tandem	IEEE	Neutral	Consistency Check Result
1 or more	0	0 or more	No message is generated and the output object file has TANDEM_FLOAT state.
0	1 or more	0 or more	No message is generated and the output object file has IEEE_FLOAT state.
1 or more	1 or more	0 or more	Error message is generated. No output object file is created.
0	0	1 or more	No message is generated and the output object file has NEUTRAL_FLOAT state.

When the floattype attribute is explicitly specified with the -set flag, TNS/R native linker sets the floattype attribute value for the output object file to that specified value. If an inconsistency is detected, a warning message and an output object file are generated. If the floating-point state is invalid, no output object file is created.

Any floating type combination is allowed if the user explicitly overrides the default with the set floattype flag.

Floating-Point State as Determined by TNS/R Native Linker floattype Attribute shows the results of each floating-point state when the floattype attribute is explicitly specified.

Table 43: Floating-Point State as Determined by TNS/R Native Linker `floattype` Attribute

Tandem	IEEE	Neutral	-set Tandem	-set IEEE	-set Neutral
1 or more	0	0 or more	No message is generated and the output object file is TANDEM_ FLOAT.	Warning message is generated and the output object file is IEEE_FLOAT.	Warning message is generated and the output object file is NEUTRAL_ FLOAT.
0	1 or more	0 or more	Warning message is generated and output object file is TANDEM_ FLOAT.	No message is generated and the output object file is IEEE_FLOAT.	Warning message is generated and the output object file is `NEUTRAL_ FLOAT` .
1 or more	1 or more	0 or more	Warning message is generated and output object file is `TANDEM_ FLOAT` .	Warning message is generated and the output object file is `IEEE_FLOAT` .	Warning message is generated and the output object file is `NEUTRAL_ FLOAT` .
0	0	1 or more	Warning message is generated and output object file is `TANDEM_ FLOAT` .	Warning message is generated and the output object file is `IEEE_FLOAT` .	No message is generated and the output object file is `NEUTRAL_ FLOAT` .

Run-Time Consistency Checking

If you attempt to use a processor that does not support IEEE floating-point format, process creation error code 64 occurs (`IEEE floating-point support not available on this processor`). Programs can call the PROCESSOR_GETINFOLIST_ procedure to determine whether a processor can run IEEE floating-point instructions. For more details about this procedure, see the *Guardian Procedure Calls Reference Manual*.

Run-time consistency checking includes a check for `floattype` consistency between the program file and the user library file, if one is used. IEEE and Tandem floating-point formats use different data formats and calling conventions: IEEE floating-point values are typically passed in IEEE floating-point registers; while Tandem floating-point values are passed in general purpose registers. Therefore, problems can occur if a function using one floating-point format calls a function using the other format. Checks are performed at process creation to ensure that the user library (if there is one) is marked with a floattype consistent with the program file.

These combinations are considered to be conflicting and are not allowed to begin to run:

floattype in program file	floattype in user library file
IEEE	Tandem
Tandem	IEEE
Neutral	IEEE

The case in which the floattype attribute of the program file is IEEE and the user library file is Neutral is not considered a conflict. In this case, the program is allowed to execute, and the C/C++ run-time libraries operate in IEEE mode.

If you are using a native user library, the library file should use the same floating-point format as the program, and the library should be marked accordingly. If the user library doesn't use floating point at all, you can mark the library NEUTRAL_FLOAT using the nld or ld -set floattype or -change floattype command. Then the user library can be used by any type of program.

Even if the user library is marked with a floattype attribute that conflicts with the program file, the program can use the library if it doesn't call anything in the user library that uses floating point. In this case, you need to mark the program file with the nld or ld -set float_lib_overrule on command to disregard the floattype attribute of the user library file.

In fact, the run-time consistency check can be overruled by using the -set float_lib_overrule on flag of the TNS/R native linker utility. If you overrule the consistency check, the operating system allows a floating-point inconsistency between the user library and the program. If you do not set float_lib_overrule, and there is an inconsistency between the program file and user library, the operating system generates an error code and does not run the program.

Linking Mixed-Language Programs

When linking mixed-language programs that use IEEE floating-point format, specify the -set floattype IEEE flag using the TNS/R native linker utility.

For example, if you have a mixed-language program composed of a C module that uses IEEE floating-point format, and a COBOL module that does not use floating point but is marked by the COBOL compiler as TANDEM_FLOAT, then you could use the ld -set floattype IEEE_FLOAT command. Or you could first use the ld -change command to change the COBOL object file to NEUTRAL_FLOAT.

This example illustrates linking a mixed-language program that uses IEEE floating-point format:

```
> ld $system.system.ccppmain cobj ptalobj -obey &
  $system.system.libcobey -set floattype ieee_float -o myexec
```

In this example, the native C object file named COBJ uses IEEE floating-point format, and the pTAL object file uses Tandem floating-point format. (Note that pTAL supports only Tandem floating-point format.) To link these modules, you must specify the -set floattype IEEE_FLOAT flag. If this flag is not specified, TNS/R native linker generates error messages because of the mismatch between Tandem and IEEE floating-point formats. When this flag is specified, ld generates a warning message about the mismatch and builds the executable file MYEXEC. CCPPMAIN (an object file) and the LIBCOBEY file (an Obey command file) are standard items required when linking C programs. To produce non-PIC code using nld, you must specify the non-PIC variant of CCPPMAIN, which is CRTLMAIN.

Use the `-set` and `-change` flags of the `ld` utility to set or change the `float_lib_overrule` attribute when creating object files. If the `float_lib_overrule` is specified more than once by either the `-set` or `-change` flags, all occurrences except the last one are ignored. The `float_lib_overrule` attribute can be changed only for executable files. An error occurs if an attempt is made to change the value of this attribute for relinkable files.

For More Information

See these manuals for more details about using IEEE floating-point format:

- *Guardian Programmer's Guide* for information about building programs using IEEE floating-point format
- *Guardian Procedure Calls Reference Manual* for information about operating mode routines and routines for converting between Tandem and IEEE floating-point formats
- The descriptions of pragmas **IEEE_FLOAT** on page 248 and **TANDEM_FLOAT** on page 319.

Working in the Guardian Environment

In the NonStop environment, you can compile and link TNS/R native programs for either the Guardian or Open System Services (OSS) environment.

Compiling a Module

The native compilers translate the source text of a module and produce:

- An extensive compiler listing. Several pragmas enable you to control the content of this compiler listing.
- A nonexecutable object file, if the compiler encountered no errors during the compilation.

After compiling all the modules that compose your program, you collect and combine them into a program file (an executable object file) by using the `nld` utility for conventional code or the `ld` utility for PIC (Position-Independent Code).

If your program comprises a single module, you can use the `RUNNABLE` pragma to direct the compiler to produce a program file instead of a nonexecutable object file.

If your program comprises more than one module, you can use the `RUNNABLE` and `LINKFILE` pragmas to direct the compiler to produce a linked program file instead of a nonexecutable object file. For more details, see pragma **LINKFILE** on page 262.

To use the `RUNNABLE` pragma, one of the modules must contain the main function of the program.

When you specify the `RUNNABLE` pragma, the native C and C++ compilers specify the $SYSTEM.SYSTEM.LIBCOBEY command file to the linker. The LIBCOBEY file directs TNS/R native linker to link to a set of standard shared run-time libraries (SRLs). For most C and C++ programs, this set of SRLs is sufficient to create an executable program. If your program requires SRLs not specified in LIBCOBEY (such as the Tools.h++ SRL) you can direct TNS/R native linker to search additional SRLs using the `LINKFILE` pragma.

The `NMC` command invokes the TNS/R native C compiler. The `NMCPLUS` command invokes the TNS/R native C++ compiler. The syntax for these commands is shown in this diagram.

```
[ RUN ] { NMC | NMCPLUS } / IN source [ , OUT listing ]

   [ , run-options ] / [ object ]
   [ ; compile-option [ , compile-option ]... ]

compile-option:

   { pragma                        }
   { define identifier [ constant ] }
   { undefine identifier             }
```

[RUN] NMC

is the TACL command to start the TNS/R native C compiler process. The RUN command keyword is optional.

[RUN] NMCPLUS

is the TACL command to start the TNS/R native C++ compiler process. The RUN command keyword is optional.

IN source

specifies the primary source file of the module. The file must be a valid Guardian file name for either a type 101 (EDIT) or type 180 disk file. Interactive input from a terminal or a process is not accepted.

OUT *listing*

specifies the file to which the TNS/R native C compiler writes the compiler listing. When specified, *listing* is usually a spooler location. If you omit the OUT option, the compiler writes the listing to your current default output file. If the file already exists, the compiler attempts to delete the file and then continue.

run-options

is a comma-separated list of additional options for the RUN command. These options are described in the *TACL Reference Manual*.

object

specifies the file to which the TNS/R native C or C++ compiler writes the object code for the source text. If you do not specify an object file, the compiler writes the object code to the file OBJECT in your current default volume and subvolume. If OBJECT cannot be created, the compiler writes the object code to the file ZZBI*nnnn* (where *nnnn* is a unique four-digit number) in your current default volume and subvolume. If you are compiling PIC, the compiler writes the object code to ZZLDF*nnn*.

compile-option

modifies compiler operation by specifying a compiler pragma or defining a preprocessor symbol.

pragma

is any valid command-line compiler pragma.

define *identifier [constant]*

defines *identifier* as a preprocessor symbol. If *identifier* is followed by a constant, *identifier* is defined as an object-like macro that expands to the given value. `define` is equivalent to using the `#define` preprocessor directive in source text.

undefine *identifier*

deletes *identifier* as a preprocessor symbol. Using undefine is equivalent to using the `#undef` preprocessor directive in source text.

Usage Guidelines

- The compiler accesses source files as text-type logical files. Consequently, the source files you specify in a module must represent physical file types that the compiler can access as text-type logical files.
- The compiler returns one of these completion codes:

0	The compilation completed successfully.
1	The compilation completed with warnings (but no errors).
2	The compilation completed with errors.
3	The compiler terminated abnormally as the result of an internal error.

Examples

1. Directing the compiler to translate the source file ABSC (which contains an entire C++ program), sending the compiler listing to the device $S.#ABSL, storing the object file under the name ABSO, and using the default VERSION3 library:

   ```
   > NMCPLUS /IN absc, OUT $s.#absl/ abso;
   ```

2. Disabling the run-time diagnostics that the `assert` function can issue by defining `NDEBUG` as a preprocessor symbol:

   ```
   > NMC /IN appc, OUT $s.#appl/ appo; define NDEBUG
   ```

3. Generating a TNS/R native object file for the OSS environment:

   ```
   > NMC /IN filec / fileo; SYSTYPE OSS
   ```

4. Generating an executable TNS/R native program from a single module:

   ```
   > NMC /IN filec / fileo; RUNNABLE
   ```

5. Generating an executable TNS/R native program composed of the modules `mod1c`, `mod2o`, `mod3o`, and `mod4o`:

   ```
   > NMC /IN mod1c / fileo; RUNNABLE, LINKFILE "myfile"
   ```

   `In this example, myfile` is a text file containing these names of object files (not source files):

   ```
   mod2o
   mod3o
   mod4o
   ```

Linking a TNS/R Module

You need to use a linker to collect and combine object files into a program file (an executable object file or a loadfile) if you did not instruct the compiler to perform the linking (that is, if you did not use the `RUNNABLE` pragma when compiling a single-module program or if you did not use the `RUNNABLE` and `LINKFILE` pragmas when compiling a multiple-module program).

This subsection provides a summary of linking C and C++ programs using the `nld` utility, the TNS/R native linker for conventional applications. For complete details about using `nld`, see the *nld Manual*.

The ld utility is the linker for PIC (Position-Independent Code). For more details about PIC and sharing code, see the:

- Pragma **CALL_SHARED** on page 210
- Pragma **NON_SHARED** on page 275
- Pragma **SHARED** on page 300
- Example of compiling and linking PIC using ld in **Examples** on page 381

For more details about ld, see the *ld Manual*.

CRTLMAIN File

The CRTLMAIN file, located in $SYSTEM.SYSTEM, contains initialization code for the TNS/R native C and C++ run-time libraries. This object file must be linked into C and C++ programs. In many instances, c89, NMC and NMCPLUS link, compile, and automatically include CRTLMAIN.

You need to explicitly link CRTLMAIN when you run the linker to create a loadfile if the linkfiles were:

- compiled without the RUNNABLE and LINKFILE options when using NMC or NMCPLUS
- compiled with the -c, -Wnolink, or -Wnomain option when using c89

The PIC (Position-Independent Code) variant of CRTLMAIN is named CCPPMAIN. On OSS environment, the PIC variant of crtlmain.o is ccppmain.o.

A similar initialization file for doing fault-tolerant programming, named CRTLNS, also might need to be linked. The PIC variant is CRTLNS2. On OSS environment, the PIC variant of crtlns.o is crtlns2.o.

Shared Run-Time Libraries (SRLs)

An SRL contains code present in virtual memory at run time, to be shared by other processes, rather than code that is linked into object files. An SRL can also contain global data, and each process using the SRL automatically gets its own run-time copy of the data, called instance data. A process can use several SRLs.

NOTE: nld supports native SRLs. It does not support the TNS shared run-time library available to TNS and accelerated programs in the OSS environment. For more details about the TNS SRL, see the *Binder Manual*.

Some of the code configured in the system library for TNS processes is packaged in SRLs for TNS/R native processes. For example, the TNS/R C run-time library, the TCP/IP sockets library, the Tools.h++ class library, and much of the OSS API are packaged as SRLs for TNS/R native processes. HPE supplies public SRLs; you cannot create your own public SRLs.

Each processor loads its public SRLs at startup from the active SYS*nn* subvolume. The SYS*nn* subvolume, located on $SYSTEM, contains a version of the operating system image for a particular node. A node can have more than one SYS*nn* subvolume, but only one active (running) SYS*nn* subvolume.

When you use nld to build an executable program, nld fixes up references to the SRLs that you have specified. You can also use nld to repeat the fix-up process on an existing program to use a new version of an SRL or let the operating system update the references when you execute the program.

To create an executable C or C++ program using nld:

1. Specify $SYSTEM.SYSTEM.CRTLMAIN (or CCPPMAIN for PIC code) to link to the TNS/R native C and C++ run-time library initialization object code file for non-PIC code.
2. Specify the object code files that you compiled.
3. Specify the shared run-time libraries (SRLs) used by your program.

   The LIBCOBEY file is a linker command file that identifies the standard set of C SRLs. Specifying – `OBEY $SYSTEM.SYSTEM.LIBCOBEY` is sufficient to link the SRLs used by most C programs. C++ programs require you to specify additional SRLs. For complete details, see **Determining Which SRLs are Required** on page 378.

4. If your program uses the active backup programming functions, such as `__ns_start_backup()`, specify the active backup programming support object code file $SYSTEM.SYSTEM.CRTLNS for non-PIC code or $SYSTEM.SYSTEM.CRTLNS2 for PIC code.

Command examples are shown in **Examples** on page 381.

SRLs and Dynamic-Link Libraries (DLLs)

The NonStop system libraries at G06.20 are compatible with the use of dynamic-link libraries (DLLs). When a NonStop server is migrated to G06.20, the system shared run-time libraries (SRLs) are automatically migrated to become hybrid SRLs, which are public libraries in PIC (Position-Independent Code) format.

You can use the `ld` utility to create a linkable object file or linkfile. You can also use `ld` to create an executable (known as a loadable object file or a loadfile) in PIC format that can function as a shared library or dynamic-link library (DLL).

Determining Which SRLs are Required

The SRLs that are required by a program depend on whether the program:

- Runs in the NonStop environment
- Uses the C run-time library
- Uses the C++ run-time library
- Uses the Tools.h++ library (and whether Version 6.1 or Version 7)
- Uses the Standard C++ Library
- Uses the TCP/IP sockets library

SRLs Available When Using VERSION1, VERSION2, and VERSION3 table is a conceptual stack of the SRLs that make up the context of `VERSION1`, `VERSION2`, and `VERSION3`. In this table, CRTL represents the SRL named `ZCRTLSRL` (the C run-time library). `RWSLSRL` is given as `RWSL` (the `VERSION2` Standard C++ Library, which is a port of the Rogue Wave standard library).

For a given version, you can link only to the SRLs listed in the column or stack for that version in **SRLs Available When Using VERSION1, VERSION2, and VERSION3**. If you attempt to link to SRLs that are listed in another column, unexpected results can occur. The order of the column or stack is also important, especially for TNS/R native C++, because it represents the order in which libraries should be loaded and accessed at run time.

For example, if you are using `VERSION1` on Guardian, you can link to ZTLHGSRL (Tools.h++ version 6), but you cannot link to `ZTLHSRL` (Tools.h++ version 7). Similarly, if you are using `VERSION3`, you cannot link to `ZRWSLSRL` (the `VERSION2` Standard C++ Library).

Note that ZSTLSRL in VERSION3 contains both the C++ run-time library and the latest Standard C++ Library. The C++ run-time library is available to VERSION1 as ZCPLGSRL or ZCPLOSRL, and to VERSION2 as ZCPLSRL, while the latest Standard C++ Library is only available to VERSION3 (the earlier version of the library is available to VERSION2 as ZRWSLSRL).

Table 44: SRLs Available When Using VERSION1, VERSION2, and VERSION3

VERSION 1		VERSION 2		VERSION 3	
Guardian	**OSS**	**Guardian**	**OSS**	**OSS**	**OSS**
TLHG	TLHO	TLH	TLH	N.A.	N.A.
N.A.	N.A.	RWSL	RWSL	STL	STL
CPLG	CPLO	CPL	CPL	STL	STL
CRTL	CRTL	CRTL	CRTL	CRTL	CRTL
CRE	CRE	CRE	CRE	CRE	CRE
OS	OSS	OS	OSS	OS	OSS
OS	OS	OS	OS	OS	OS

Notes: SRL names are shown without the leading Z and these SRL; therefore, STL represents ZSTLSRL (the combined C++ run-time library and the C++ Standard Library for VERSION3), and TLH represents ZTLHSRL (the Tools.h++ library for VERSION2).

OS represents the operating system. OSS represents the Open System Services environment.

CRE represents the common run-time environment.

For more details, see:

- **Using the Standard C++ Library** on page 88
- Pragmas **VERSION1** on page 322 , **VERSION2** on page 324, and **VERSION3** on page 326
- **C++ Run-Time Library and Standard C++ Library** on page 29

Additional SRLs might be required for other run-time libraries and services. For example, if you use the VERSION2 pragma with the NMCPLUS driver in the Guardian environment (or the -Wcplusplus and –Wversion2 flags with the c89 driver in the OSS environment), these SRLs are automatically searched:

- ZCPLSRL (the C++ run-time SRL)
- ZRWSLSRL (the VERSION2 Standard C++ Library SRL)
- The SRLs in LIBCOBEY

If, however, you invoke TNS/R native linker explicitly, you need to include these SRLs in the link command using the -lib option, as shown in **Using the Guardian TNS/R Native Linker Utilities to Link SRLs** table.

Table 45: Using the Guardian TNS/R Native Linker Utilities to Link SRLs

If your program uses:	You should specify these TNS/R native linker utility flags:

C run-time library and runs in the Guardian environment	`-OBEY $SYSTEM.SYSTEM.LIBCOBEY`
	or
	`-l ZCRTLSRL  -l ZCRESRL`

C run-time library and runs in the OSS environment	`-OBEY $SYSTEM.SYSTEM.LIBCOBEY`
	or
	`-l ZCRTLSRL  -l ZCRESRL  -l ZOSSKSRL`
	`-l ZOSSFSRL  -l ZSECSRL  -l ZI18NSRL`
	`-l ZICNVSRL  -l ZOSSESRL -l ZINETSRL`
	`-l ZOSSHSRL  -l ZSTFNSRL`

VERSION2 C++ run-time library and runs in the Guardian environment	`-l ZRWSLSRL  -l ZCPLSRL`
	`-OBEY $SYSTEM.SYSTEM.LIBCOBEY`
	or
	`-l ZRWSLSRL  -l ZCPLSRL  -l ZCRTLSRL`
	`-l ZCRESRL`

VERSION2 C++ run-time library and runs in the OSS environment	`-l ZRWSLSRL  -l ZCPLSRL   -l ZOSSHSRL`
	`-l ZSTFNSRL`
	`-OBEY $SYSTEM.SYSTEM.LIBCOBEY`
	or
	`-l ZRWSLSRL  -l ZCPLSRL   -l ZCRTLSRL`
	`-l ZCRESRL   -l ZOSSKSRL  -l ZOSSFSRL`
	`-l ZSECSRL   -l ZI18NSRL  -l ZICNVSRL`
	`-l ZOSSESRL  -l ZINETSRL  -l ZOSSHSRL`
	`-l ZSTFNSRL`

VERSION2 Standard C++ Library, Tools.h++ (version 7) and runs in the Guardian environment	`-l ZTLHSRL   -l ZRWSLSRL  -l ZCPLSRL`
	` -OBEY $SYSTEM.SYSTEM.LIBCOBEY`
	or
	`-l ZTLHSRL   -l ZRWSLSRL  -l ZCPLSRL`
	`-l ZCRTLSRL  -l ZCRESRL`

VERSION2 Standard C++ Library, Tools.h++ (version 7) and runs in the OSS environment	`-l ZTLHSRL   -l ZRWSLSRL  -l ZCPLSRL`
	`-OBEY $SYSTEM.SYSTEM.LIBCOBEY`
	or
	`-l ZTLHSRL   -l ZRWSLSRL  -l ZCPLSRL`
	`-l ZOSSHSRL  -l ZCRTLSRL  -l ZCRESRL`
	`-l ZOSSKSRL  -l ZOSSFSRL  -l ZSECSRL`
	`-l ZI18NSRL  -l ZICNVSRL  -l ZOSSESRL`
	`-l ZINETSRL  -l ZSTFNSRL`

Table Continued

If your program uses:	You should specify these TNS/R native linker utility flags:

VERSION3 Standard C++ Library and runs in the OSS or Guardian environment	`-l ZSTLSRL`
	`-OBEY $SYSTEM.SYSTEM.LIBCOBEY`

OSS `nlist()` function	`-l ZUTILSRL` and other SRLs required by the program environment
TCP/IP socket library	`-l ZINETSRL` and other SRLs required by the program environment

For more details about the linkers, see these manuals:

- *nld Manual*
- *ld Manual*

Examples

1. The specified `nld` flags link a `VERSION2` Guardian C++ program that uses the Tools.h++ class libraries (ZTLHSRL) and the Standard C++ Library (ZRWSLSRL):

```
    > NLD $SYSTEM.SYSTEM.CRTLMAIN MYOBJ -o MYEXEC   &
      -l ZTLHSRL  -l ZRWSLSRL  -l ZCPLSRL -l ZCRTLSRL &
      -l ZCRESRL
```

2. The specified `nld` flags link a `VERSION2` OSS C program:

```
> NLD $SYSTEM.SYSTEM.CRTLMAIN MYOBJ -o MYEXEC  &
     -l ZTLHSRL   -l ZRWSLSRL  -l ZCPLSRL &
     -l ZOSSHSRL  -l ZCRTLSRL  -l ZCRESRL &
     -l ZOSSKSRL  -l ZOSSFSRL  -l ZSECSRL &
     -l ZI18NSRL  -l ZICNVSRL  -l ZOSSESRL &
     -l ZINETSRL  -l ZSTFNSRL
```

3. A simplified version of the previous example:

```
> NLD $SYSTEM.SYSTEM.CRTLMAIN MYOBJ -o MYEXEC &
      -OBEY $SYSTEM.SYSTEM.LIBCOBEY &
      -l ZTLHSRL -l ZRWSLSRL -l ZCPLSRL
```

4. Linking with the `ld` linker and the `VERSION3` Standard C++ Library (the default beginning library beginning with G06.20):

```
> LD $SYSTEM.SYSTEM.CCPPMAIN MYOBJ -o MYEXEC &
         -OBEY $SYSTEM.SYSTEM.LIBCOBEY -l ZSTLSRL
```

5. Compiling PIC (Position-Independent Code) using the default native C++ dialect (`VERSION3`). The example program (MEXE) uses a DLL (named NDLL) compiled from a library file named NC, which contains the `getnum()` function. MEXE imports `getnum()` and prints the result (31).

Note the use of the `import$` and `export$` keywords, the `SHARED` pragma (to compile the library) and `CALL_SHARED` pragma (to compile the main module), and the `ld` and `rld` utilities (the PIC linker and loader). The result is a dynamic-link library (DLL) named NDLL:

Source file (named MC):

```
import$ extern int getnum();

int main()
{
  int x = -99;
  x = getnum();
  printf (“x was -99; is now %d”, x);
  return 0;
}
```

Library for DLL (file name NC):

```
export$ int getnum()
{
  return 31;
}
```

Compiler and Linker Commands:

```
nmc / in NC, out NLST / NDLL; shared
== Compile NC with shared (as a DLL); linkfile is PIC

nmc / in MC, out MLST / MOBJ; call_shared
== Compile MC module with call_shared; linkfile is PIC

ld / out LLST / mobj $SYSTEM.SYSTEM.CCPPMAIN  -obey &
  $SYSTEM.SYSTEM.LIBCOBEY -libvol $myvol.svol -lib NDLL &
  -o MEXE

== Build MEXE, specifying CCPPMAIN (CRE component).
== LIBCOBEY specifies standard SRLs to be searched.
```

Compiling and Linking TNS/E Native C and C++ Programs

The TNS/E native C and C++ compilers take as input a module (a translation unit) and generate an object file or linkfile. A module is defined as a source file with all the headers and source files it includes, except for any source lines skipped as the result of conditional preprocessor directives.

The `eld` utility is the TNS/E native linker that links PIC (Position-Independent Code) linkfiles to produce PIC loadfiles.

The SQL compiler processes executable files and generates code for embedded SQL statements.

The native `c89` and `c99` utilities control the TNS/E native C and C++ compilation system in the Open System Services (OSS) environment. Note that the `c89` utility on TNS/E systems can generate TNS/R code for linking by the TNS/R native linker utility (not the `eld` utility) when the `-Wtarget=tns/r` flag is used; discussions of OSS `c89` in **Compiling and Linking TNS/R Native C and C++ Programs** on page 367 therefore apply to TNS/E users when that flag is specified.

For information on how to compile and link programs in the OSS environment, see the `c89(1)` or `c99(1)` reference page online or in the *Open System Services Shell and Utilities Reference Manual*.

For information about compiling and binding native C++ programs using the Windows PC environment, see the online help for the HPE Enterprise Tool Kit—NonStop Edition (ETK). In this guide, the Enterprise Tool Kit is introduced in **Using the Native C/C++ Cross Compiler on the PC** on page 419.

For information about compiling and binding native C++ programs using NSDEE, see the online help for NSDEE.

The Guardian CPPINIT* and OSS or PC `cppinit`* files used to create, replace, or delete C++ operators for TNS/R programs are not needed for TNS/E programs. For TNS/E, the files LIBCTXT and libc.txt can be used for TNS/E linking. (This is similar to how Guardian LIBCOBEY and OSS libc.obey files are used for TNS/R linking).

Selecting a Development Platform

A development platform consists of the hardware system and software environment available to compile, link, and run a program.

You can develop OSS programs regardless of whether the OSS environment is available on the system. However, you cannot run and test OSS programs on a system without the OSS environment.

It is easier to develop a program in the environment in which it runs, but you can develop a program in one environment that runs in another environment, with a few restrictions. However, compile times are much faster in the PC environment.

Development Platform Capabilities (TNS/E Native C and C++ Programs) table describes the capabilities of each development platform.

Table 46: Development Platform Capabilities (TNS/E Native C and C++ Programs)

Capability	System With Guardian Environment	System With Guardian and OSS Environments	ETK on Windows PC	NSDEE on Windows PC
Use Guardian development tools for Guardian programs?	Yes	Yes	No	Yes
Use Guardian development tools for OSS programs?	Yes	Yes	No	Yes
Use OSS development tools for Guardian programs?	No	Yes	No	Yes
Use OSS development tools for OSS programs?	No	Yes	No	Yes
Run Guardian programs?	Yes	Yes	No	Yes
Run OSS programs?	No	Yes	No	Yes
Compile programs with embedded SQL?	Yes	Yes	Yes	Yes
Use PC-based development tools for Guardian and OSS programs?	No	No	Yes	Yes

These restrictions apply to developing Guardian programs with OSS tools:

- You cannot use the `RUNNABLE` and `SEARCH` pragmas. However, you can direct the `c89` utility to bind implicitly after a compilation. You can also specify library files to be searched using the `c89 -L` flag.
- You cannot use the `SSV` pragma. However, you can specify search directories using the `c89 -I` flag.

Specifying Header Files

The macros and functions in the TNS/E native C run-time library are declared in header files. Each header file contains declarations for a related set of library functions and macros, in addition to variables and types that complete that set. If you use a function in the library, you should include the header file in which it is declared. You should not declare the routine yourself because the declarations in the header files have provisions for several situations that can affect the form of a given declaration, including:

- Whether the routine is implemented internally as a function or a macro
- Whether the function is written in a language other than C
- Whether you are compiling for the small-memory or large-memory model
- Whether you are compiling for the 16-bit or 32-bit (wide) data model
- Whether you are compiling for the NonStop environment
- Whether you are compiling for TNS mode or native mode

In addition, the header file prototype declarations enable the compiler to check parameters and arguments for compatibility, ensuring that function calls provide the correct number and type of arguments.

A single set of TNS/E native C library header files supports all three environments (Guardian, OSS, and PC) and all three modes (TNS, TNS/R native, and TNS/E native). The locations of the header files in the three environments is summarized in **Locations of Header Files** table.

Table 47: Locations of Header Files

Environment	Location
Guardian	$SYSTEM.SYSTEM and $SYSTEM.ZTCPIP
OSS	`/usr/include` and its subdirectories
PC	ETK Version 3.0 or later: `\Compaq ETK-NSE\rel\usr\include,` where *rel* represents the release version update, such as H06.02

To specify header files, use the `#include` preprocessor directive. For example, to include the STDIO header file, specify in your object file:

```
#include <stdio.h>
```

You might also need to specify header files for the TNS/E native Standard C++ Library and Tools.h++ library to use functions contained in those libraries. For examples of `#include` directives for these libraries, see **Using the Standard C++ Library** on page 88 and **Accessing Middleware Using HPE C and C++ for NonStop Systems** on page 104. For information about including DLLs at link time, see **Determining Which SRLs are Required** on page 378.

You can specify locations to search for header files:

- In the Guardian environment, use the `SSV` pragma to specify a search list of subvolumes for files specified in `#include` directives. For more details, see the pragma **SSV** on page 308.
- In the OSS environment, use the `-I` flag to the `c89` utility to specify a search list of directories for files specified in # `include` directives. For more details, see the `c89(1)` reference page either online or in the *Open System Services Shell and Utilities Reference Manual*.

- In the PC (Enterprise Tool Kit) environment, specify a search list of directories using the Directories page.
- In the PC command line environment (using the cross compilers), use the `-I` flag to the `c89` utility to specify a search list of directories for files specified in `#include` directives. For more details, see the document *Using the Command-Line Cross Compilers on Windows*.

While header files are optional (but strongly recommended) for programs that contain Guardian or OSS modules exclusively, header files are required for mixed-module programs. If you do not compile using header files, `eld` cannot correctly resolve external references to Guardian and OSS versions of C functions.

Compiling and Linking Floating-Point Programs

You can now choose either Tandem floating-point format or IEEE floating-point format for performing floating-point arithmetic in your native C and C++ programs. This table compares the two formats.

Tandem Floating-Point Format	IEEE Floating-Point Format
Default for TNS/R modules	Default for TNS/E modules
A proprietary implementation of floating-point arithmetic that is supported in software millicode	An industry-standard data format that is supported in the processor hardware
Provides backward compatibility with pre-G06.06 C and C++ applications	Requires the G06.06 C and C++ applications and the G06.06 or later RVUs for the TNS/R native C and C++ compilers; requires the H06.03 or later RVUs for the TNS/E native C and C++ compilers.
Available for TNS C and C++, FORTRAN, TAL, pTAL, D-series Pascal, COBOL, and native C and C++ programs	Available only for native C and C++ programs. Used internally by TNS/E native COBOL programs
Requires conversion routines for data interchange between Tandem format and IEEE format (see the *Guardian Procedure Calls Reference Manual* for more detail about conversion routines)	Provides easier data interchange with other systems using 64-bit and 32-bit IEEE floating-point data formats; interchange typically consists of reversing the byte order to convert between big-endian data format (on NonStop systems) and little-endian format (on the target system)
	Provides better handling of exception conditions such as overflow and underflow; the existence of NaN (not a number), infinities, and exception flags make it easier to detect invalid results

Using Compiler Pragmas IEEE_Float and Tandem_Float

- To use IEEE floating-point format, you can optionally specify the IEEE_FLOAT pragma on the command line when running the TNS/E native C or C++ compiler. IEEE_FLOAT is the default setting for TNS/E native compilations.

If you are using TNS/E native C++, you also need to specify the `VERSION2` or `VERSION3` directive. For more details, see the pragmas **IEEE_FLOAT** on page 248, **VERSION2** on page 324, and **VERSION3** on page 326.

- To use Tandem floating-point format, you must specify the TANDEM_FLOAT pragma on the command line when running the TNS/E native C or C++ compiler.

  TANDEM_FLOAT is the default setting for TNS/R native compilations. See the pragma **TANDEM_FLOAT** on page 319.

The native compilers set the floating-point format type in the generated object file.

Two examples of compiling native C and C++ programs that use IEEE floating-point format:

```
> CCOMP / IN SOURCEA, OUT $.#LIST / OBJECTA; IEEE_FLOAT


> CPPCOMP / IN SOURCEB, OUT $S.#LIST / OBJECTB; VERSION2, &
  IEEE_FLOAT
```

Using Link Options to Specify Floating-Point Format

When linking object files using the `eld` utility, you can specify the floating-point state of the output object file using the `-set floattype` flag. You can set any of these options:

```
TANDEM_FLOAT
IEEE_FLOAT
NEUTRAL_FLOAT
```

If the -set `floattype` flag is not specified, the eld utility derives the floating-point state of the output object file from the states of the input files.

When modifying an existing object file, eld sets the state as specified by the –change `floattype` flag.

Link-Time Consistency Checking

The eld utility checks the consistency of floating-point type combinations when linking object files. The checking differs according to whether the `-set` flag is specified.

When the `floattype` attribute is not explicitly set with the `-set` flag, `eld` uses the `floattype` attribute values of all the input object files for determining the `floattype` attribute value for the output object file. If the consistency checks of the input object files result in an invalid floating-point state or inconsistent value, an error message is generated and no output object file is created.

Floating Point Consistency Check by eld Utility table shows the results of each combination of floating-point states when the `floattype` attribute is not explicitly specified. The columns Tandem, IEEE, and Neutral, show the number of input object files characterized by that floating-point setting.

Table 48: Floating Point Consistency Check by eld Utility

Tandem	IEEE	Neutral	Consistency Check Result
1 or more	0	0 or more	No message is generated and the output object file has TANDEM_FLOAT state.
0	1 or more	0 or more	No message is generated and the output object file has IEEE_FLOAT state.

Table Continued

Tandem	IEEE	Neutral	Consistency Check Result
1 or more	1 or more	0 or more	Error message is generated. No output object file is created.
0	0	1 or more	No message is generated and the output object file has NEUTRAL_FLOAT state.

When the `floattype` attribute is explicitly specified with the `-set` flag, `eld` sets the `floattype` attribute value for the output object file to that specified value. If an inconsistency is detected, a warning message and an output object file are generated. If the floating-point state is invalid, no output object file is created.

Any floating type combination is allowed if the user explicitly overrides the default with the set `floattype` flag.

Floating-Point State as Determined by `eld floattype` Attribute table shows the results of each floating-point state when the `floattype` attribute is explicitly specified.

Table 49: Floating-Point State as Determined by `eld floattype` Attribute

Tandem	IEEE	Neutral	-set Tandem	-set IEEE	-set Neutral
1 or more	0	0 or more	No message is generated and the output object file is `TANDEM_FLOAT` .	Warning message is generated and the output object file is `IEEE_FLOAT` .	Warning message is generated and the output object file is `NEUTRAL_FLOAT` .
0	1 or more	0 or more	Warning message is generated and output object file is `TANDEM_FLOAT` .	No message is generated and the output object file is `IEEE_FLOAT` .	Warning message is generated and the output object file is `NEUTRAL_FLOAT` .
1 or more	1 or more	0 or more	Warning message is generated and output object file is `TANDEM_FLOAT` .	Warning message is generated and the output object file is `IEEE_FLOAT` .	Warning message is generated and the output object file is `NEUTRAL_FLOAT` .
0	0	1 or more	Warning message is generated and output object file is `TANDEM_FLOAT` .	Warning message is generated and the output object file is `IEEE_FLOAT` .	No message is generated and the output object file is `NEUTRAL_FLOAT` .

Run-Time Consistency Checking

Run-time consistency checking includes a check for `floattype` consistency between the program file and the user library file, if one is used. IEEE and Tandem floating-point formats use different data formats and calling conventions: IEEE floating-point values are typically passed in IEEE floating-point registers; while Tandem floating-point values are passed in general-purpose registers. Therefore, problems can occur if a function using one floating-point format calls a function using the other format. Checks are performed at process creation to ensure that the user library (if there is one) is marked with a floattype consistent with the program file.

These combinations are considered to be conflicting and are not allowed to begin to run:

floattype in program file	floattype in user library file
IEEE	Tandem
Tandem	IEEE
Neutral	IEEE

The case in which the floattype attribute of the program file is IEEE and the user library file is Neutral is not considered a conflict. In this case, the program is allowed to execute, and the C/C++ run-time libraries operate in IEEE mode.

If you are using a native user library, the library file should use the same floating-point format as the program, and the library should be marked accordingly. If the user library doesn't use floating point at all, you can mark the library NEUTRAL_FLOAT using the eld -set floattype or -change floattype command. Then the user library can be used by any type of program.

Even if the user library is marked with a floattype attribute that conflicts with the program file, the program can use the library if it does not call anything in the user library that uses floating point. In this case, you need to mark the program file with the eld -set float_lib_overrule on command to disregard the floattype attribute of the user library file.

In fact, the run-time consistency check can be overruled by using the -set float_lib_overrule on flag of the eld utility. If you overrule the consistency check, the operating system allows a floating-point inconsistency between the user library and the program. If you do not set float_lib_overrule, and there is an inconsistency between the program file and user library, the operating system generates an error code and does not run the program.

Linking Mixed-Language Programs

When linking mixed-language programs that use IEEE floating-point format, specify the -set floattype IEEE flag using the eld utility.

For example, if you have a mixed-language program composed of a C module that uses IEEE floating-point format, and a COBOL module that doesn't use floating point but is marked by the native COBOL compiler as TANDEM_FLOAT, then you could use the eld -set floattype IEEE_FLOAT command. Or you could first use the eld -change command to change the COBOL object file to NEUTRAL_FLOAT.

This example illustrates linking a mixed-language program that uses IEEE floating-point format:

```
> ELD $SYSTEM.SYSTEM.CCPLMAIN COBJ EPTALOBJ &
  -set floattype ieee_float -o MYEXEC
```

In this example, the native C object file named COBJ uses IEEE floating-point format, and the EPTAL-created object file named EPTALOBJ uses Tandem floating-point format. (Note that EPTAL supports only Tandem floating-point format.) To link these modules, you must specify the -set floattype IEEE_FLOAT flag. If this flag is not specified, eld generates error messages because of the mismatch between Tandem and IEEE floating-point formats. When this flag is specified, eld generates a warning message about the mismatch and builds the executable file MYEXEC. CCPLMAIN (an object file) is a standard item, required when linking C programs.

Use the `-set` and `-change` flags of the `eld` utility to set or change the `float_lib_overrule` attribute when creating object files. If the `float_lib_overrule` is specified more than once by either the `-set` or `-change` flags, all occurrences except the last one are ignored. The `float_lib_overrule` attribute can be changed only for executable files. An error occurs if an attempt is made to change the value of this attribute for relinkable files.

For More Information

See these manual for more details about using floating-point formats:

- *Guardian Programmer's Guide* for information about building programs using IEEE floating-point format
- *Guardian Procedure Calls Reference Manual* for information about operating mode routines and routines for converting between Tandem and IEEE floating-point formats
- The descriptions of pragmas **IEEE_FLOAT** on page 248 and **TANDEM_FLOAT** on page 319.

Working in the Guardian Environment

In the Guardian environment, you can compile and link programs for either the Guardian or Open System Services (OSS) environment.

Compiling a Module

The native compilers translate the source text of a module and produce:

- An extensive compiler listing; several pragmas enable you to control the content of this compiler listing
- A nonexecutable object file (if the compiler encountered no errors during the compilation)

After compiling all the modules that compose your program, you collect and combine them into a program file (an executable object file) by using the `eld` utility.

If your program comprises a single module, you can use the `RUNNABLE` pragma to direct the compiler to produce a program file instead of a nonexecutable object file.

If your program comprises more than one module, you can use the `RUNNABLE` and `LINKFILE` pragmas to direct the compiler to produce a linked program file instead of a nonexecutable object file. For more details, see pragma **LINKFILE** on page 262.

To use the `RUNNABLE` pragma, one of the modules must contain the main function of the program.

The CCOMP command invokes the TNS/E native C compiler. The CPPCOMP command invokes the TNS/E native C++ compiler. The syntax for these commands is shown in this diagram.

```
[ RUN ] { CCOMP | CPPCOMP } / IN source [ , OUT listing ]
      [ , run-options ] / [ object ]
      [ ; compile-option [ , compile-option ]... ]

compile-option:
      { pragma                                       }
      { define identifier [ constant ] }
      { undefine identifier                      }
```

[RUN] CCOMP

is the TACL command to start the TNS/E native C compiler process. The RUN command keyword is optional.

[RUN] CPPCOMP

is the TACL command to start the TNS/E native C++ compiler process. The RUN command keyword is optional.

IN *source*

specifies the primary source file of the module. The file must be a valid Guardian file name for either a type 101 (EDIT) or type 180 disk file. Interactive input from a terminal or a process is not accepted.

OUT *listing*

specifies the file to which the native C compiler writes the compiler listing. When specified, *listing* is usually a spooler location. If you omit the OUT option, the compiler writes the listing to your current default output file. If the file already exists, the compiler attempts to delete the file and continue.

run-options

is a comma-separated list of additional options for the RUN command. These options are described in the *TACL Reference Manual*.

object

specifies the file to which the TNS/E native C or C++ compiler writes the object code for the source text. If you do not specify an object file, the compiler writes the object code to the file OBJECT in your current default volume and subvolume. If OBJECT cannot be created, the compiler writes the object code to the file ZZLDF*nnn* (where *nnn* is a unique three-digit number) in your current default volume and subvolume.

compile-option

modifies compiler operation by specifying a compiler pragma or defining a preprocessor symbol.

pragma

is any valid command-line compiler pragma.

define *identifier [constant]*

defines *identifier* as a preprocessor symbol. If *identifier* is followed by a constant, *identifier* is defined as an object-like macro that expands to the given value. `define` is equivalent to using the `#define` preprocessor directive in source text.

undefine *identifier*

deletes *identifier* as a preprocessor symbol. Using undefine is equivalent to using the `#undef` preprocessor directive in source text.

Usage Guidelines

- The compiler accesses source files as text-type logical files. Consequently, the source files you specify in a module must represent physical file types that the compiler can access as text-type logical files.
- The compiler returns one of these completion codes:

0	The compilation completed successfully.
1	The compilation completed with warnings (but no errors).
2	The compilation completed with errors.
3	The compiler terminated abnormally as the result of an internal error.

Examples

1. Directing the compiler to translate the source file ABSC (which contains an entire C++ program), sending the compiler listing to the device $S.#ABSL, storing the object file under the name ABSO, and using the default VERSION3 library:

   ```
   > CPPCOMP /IN absc, OUT $s.#absl/ abso;
   ```

2. Disabling the run-time diagnostics that the `assert` function can issue by defining `NDEBUG` as a preprocessor symbol:

   ```
   > CCOMP /IN appc, OUT $s.#appl/ appo; define NDEBUG
   ```

3. Generating an object file for the OSS environment:

   ```
   > CCOMP /IN filec / fileo; SYSTYPE OSS
   ```

4. Generating an executable program from a single module:

   ```
   > CCOMP /IN filec / fileo; RUNNABLE
   ```

5. Generating an executable program composed of the modules `mod1c`, `mod2o`, `mod3o`, and `mod4o`:

   ```
   > CCOMP /IN mod1c / fileo; RUNNABLE, LINKFILE "myfile"
   ```

   In this example, `myfile` is a text file containing these names of object files (not source files):

   ```
   mod2o
   mod3o
   mod4o
   ```

Linking a TNS/E Module

You need to use a linker to collect and combine object files into a program file (an executable object file or a loadfile) if you did not instruct the compiler to perform the linking (that is, if you did not use the `RUNNABLE` pragma when compiling a single-module program or if you did not use the `RUNNABLE` and `LINKFILE` pragmas when compiling a multiple-module program).

This subsection provides a summary of linking TNS/E native C and C++ programs using the `eld` utility, the TNS/E native linker. The `eld` utility is a linker for PIC (Position-Independent Code). For more details about PIC and sharing code, see:

- Pragma **CALL_SHARED** on page 210
- Pragma **SHARED** on page 300
- Example of compiling and linking PIC using `eld`

  in **Examples** on page 381

For more details about `eld`, see the *eld Manual*.

CCPLMAIN File

The CCPLMAIN file, located in $SYSTEM.SYSTEM, contains initialization code for the TNS/E native C and C++ run-time libraries. This object file must be linked into C and C++ programs. In many instances, `c89`, CCOMP and CPPCOMP link, compile, and automatically include CCPLMAIN.

You need to explicitly link CCPLMAIN when you run the linker to create a loadfile if the linkfiles were:

- Compiled without the `RUNNABLE` and `LINKFILE` options when using CCOMP or CPPCOMP
- Compiled with the `-c`, `-Wnolink`, or `-Wnomain` option when using `c89`

A similar initialization file for doing fault-tolerant programming, named CRTLNSE, also might need to be linked. On OSS environment, the file is named `crtlnse.o`.

NOTE: For programs compiled with the LP64 data model, the name of the main program file is CMAIN64.

Dynamic-Link Libraries (DLLs)

A DLL contains code present in virtual memory at run time, to be shared by other processes, rather than code that is linked into object files. A DLL can also contain global data, and each process using the DLL automatically gets its own run-time copy of the data, called instance data. A process can use several DLLs.

HPE supplies public DLLs; you can create your own public DLLs.

Each process loads its public DLLs at startup from the active ZDLL*nnn* subvolume. The ZDLL*nnn* subvolume, located on $SYSTEM, contains a version of the public DLLs supplied by HPE. A node can have more than one ZDLL*nnn* subvolume, but only one active (running) ZDLL*nnn* subvolume.

Some code is automatically provided from the implicit libraries in the SYS*nn* subvolume. The SYS*nn* subvolume, located on $SYSTEM, contains a version of the operating system files for a particular node. You do not need to link code kept in the implicit libraries.

When you use `eld` to build an executable program, `eld` fixes up references to the DLLs that you have specified. You can also use `eld` to repeat the fix-up process on an existing program to use a new version of an DLL or let the `rld` run-time loader or operating system update the references when you execute the program.

To create an executable TNS/E native C or C++ program using `eld`:

1. Specify $SYSTEM.SYSTEM.CCPLMAIN to link to the TNS/E native C and C++ run-time library initialization object code file.
2. Specify the object code files that you compiled.
3. Specify the DLLs used by your program. For complete details, see **Determining Which SRLs are Required** on page 378.
4. If your program uses the active backup programming functions, such as `__ns_start_backup()`, specify the active backup programming support object code file $SYSTEM.SYSTEM.CRTLNSE.

Command examples are shown in **Examples** on page 381.

Shared Run-Time Libraries and Dynamic-Link Libraries (DLLs)

The NonStop system implicit libraries are compatible with the use of dynamic-link libraries (DLLs). The system implicit DLLs are public libraries in PIC (Position-Independent Code) format.

You use the `eld` utility to create a linkable object file or linkfile. You can also use `eld` to create an executable (known as a loadable object file or a loadfile) in PIC format that can function as a dynamic-link library (DLL).

Shared run-time libraries are not supported for TNS/E code.

Determining Which DLLs are Required

The DLLs that are required by a program depend on whether the program:

- Runs in the NonStop environment
- Uses the C run-time library
- Uses the C++ run-time library
- Uses the Tools.h++ library (and whether Version 6.1 or Version 7)
- Uses the Standard C++ Library
- Uses the TCP/IP sockets library

The following table is a conceptual stack of the DLLs that make up the context of `VERSION2` and `VERSION3` . In this table, CPPC represents the DLL named `ZCPPCDLL` (the C run-time library). `ZCPP2DLL` is given as `CPP2` (the `VERSION2` Standard C++ Library).

For a given version, you can link only to the DLLs listed in the column or stack for that version in the following table. If you attempt to link to DLLs that are listed in another column, unexpected results can occur. The order of the column or stack is also important, especially for TNS/E native C++, because it represents the order in which libraries should be loaded and accessed at run time.

For example, if you are using `VERSION2` on Guardian, you can link to `ZTLH7DLL` (Tools.h++ version 6), but you cannot link to `ZCPP3DLL` (Tools.h++ version 7). Similarly, if you are using `VERSION3` , you cannot link to `ZCPP2DLL` (the `VERSION2` Standard C++ Library).

Table 50: DLLs Available When Using VERSION2 and VERSION3

VERSION 2		VERSION 3	
Guardian	**OSS**	**Guardian**	**OSS**
TLH7	TLH7	N.A.	N.A.
CPP2	CPP2	CPP3	CPP3
	CPPC	CPPC	
	CRTL	CRTL	
	CRE	CRE	
OS	OSS	OS	OSS

Table Continued

VERSION 2		VERSION 3	
Guardian	OSS	Guardian	OSS
OS	OS	OS	OS

Notes: DLL names are shown without the leading Z and these DLL; therefore, CPPC represents ZCPPCDLL, and TLH7 represents ZTLH7DLL.

OS represents the HPE NonStop OS.

OSS represents the Open System Services environment.

CRTL represents the C run-time library. CRE represents the common run-time environment. For other environments such as the PC, the names are prefixed with `lib` and suffixed with `.so` ; therefore, CPPC is `libcppc.so`.

For more details, see:

- **Using the Standard C++ Library** on page 88
- Pragmas **VERSION2** on page 93, **VERSION3** on page 92, and **VERSION4** on page 91
- **C++ Run-Time Library and Standard C++ Library** on page 29

Additional DLLs might be required for other run-time libraries and services. For example, if you use the `VERSION2` pragma with the CPPCOMP driver in the Guardian environment (or the `-Wcplusplus` and `-Wversion2` flags with the `c89` driver in the OSS environment), these DLLs are automatically searched:

- ZCPPCDLL (the common C++ run-time DLL)
- ZCPP2DLL (the `VERSION2` Standard C++ library DLL)

If, however, you invoke `eld` explicitly, you need to include these DLLs in the link command using the `-lib` option, as shown in the following table.

Table 51: Using the Guardian eld Utility to Link DLLs

If your program uses:	You should specify these eld utility flags:
C run-time library and runs in the Guardian environment	`-l CRTL -l CRED`
C run-time library and runs in the OSS environment	`-l CRTL -l CRE -l OSSK -l OSSF -l SEC -l I18N -l ICNV -l OSSE -l INET -l OSSH`
`VERSION2` C++ run-time library and runs in the Guardian environment	`-l CPP2 -l CPPC -l CRTL -l CRE`
`VERSION2` C++ run-time library and runs in the OSS environment	`-l CPPC -l CPP2 -l CRTL -l CRE -l OSSK -l OSSF -l SEC -l I18N -l ICNV -l OSSE -l INET -l OSSH`

Table Continued

If your program uses:	You should specify these eld utility flags:
VERSION2 Standard C++ Library, Tools.h++ (version 7) and runs in the Guardian environment	-l TLH7 -l CPP2 -l CPPC -l CRTL -l CRE
VERSION2 Standard C++ Library, Tools.h++ (version 7) and runs in the OSS environment	-l TLH7 -l CPP2 -l CPPC -l OSSH -l CRTL -l CRE -l OSSK -l OSSF -l SEC -l I18N -l ICNV -l OSSE -l INET
VERSION3 Standard C++ Library and runs in the OSS or Guardian environment	-l CPPC -l CPP3
OSS nlist() function	-l UTIL and other DLLs required by the program environment
TCP/IP socket library	-l INET and other DLLs required by the program environment

For more details about the linker, see these manuals:

- *eld Manual*
- *rld Manual*

Examples

1. The specified eld flags link a VERSION2 Guardian C++ program that uses the Tools.h++ class libraries (TLH7) and the Standard C++ Library (CPP2):

```
> ELD $SYSTEM.SYSTEM.CCPLMAIN MYOBJ -o MYEXEC   &
      -l TLH7  -l CPPC  -l CPP2 -l CRTL &
      -l CRE
```

2. The specified eld flags link a VERSION2 OSS C program:

```
> ELD $SYSTEM.SYSTEM.CCPLMAIN MYOBJ -o MYEXEC &
     -set systype oss &
     -l TLH7 -l CPPC -l CPP2 &
     -l OSSH -l CRTL -l CRE &
     -l OSSK -l OSSF -l SEC &
     -l I18N -l ICNV -l OSSE &
     -l INET
```

3. Linking with the eld linker and the VERSION3 Standard C++ Library (the default library):

```
> ELD $SYSTEM.SYSTEM.CCPLMAIN MYOBJ -o MYEXEC &
     -l CPP3 -l CPPC
```

4. The specified eld flags link a LP64 OSS C program:

```
> ELD $SYSTEM.SYSTEM.CMAIN64 MYOBJ -o MYEXEC &
          -set systype oss &
```

```
        -l CRTl -l CRE -l OSSK -l OSSF -l SEC &
        -l I18N -l ICNV -l OSSE -l INET -l OSSH
```

5. Compiling PIC (Position-Independent Code) using the default TNS/E native C++ dialect (`VERSION3`). The example program (MEXE) uses a DLL (named NDLL) compiled from a library file named NC, which contains the `getnum()` function. MEXE imports `getnum()` and prints the result (31).

   Note the use of the `import$` and `export$` keywords, the `SHARED` pragma (to compile the library) and `CALL_SHARED` pragma (to compile the main module), and the `eld` and `rld` utilities (the PIC linker and loader). The result is a dynamic-link library (DLL) named NDLL:

   Source file (named MC):

```
import$ extern int getnum();

int main()
{
  int x = -99;
  x = getnum();
  printf (“x was -99; is now %d”, x);
  return 0;
}
Library for DLL (file name NC):

export$ int getnum()
{
  return 31;
}
```

   Compiler and Linker Commands:

```
CCOMP / in NC, out NLST / NDLL; shared
== Compile NC with shared (as a DLL); linkfile is PIC

CCOMP / in MC, out MLST / MOBJ; call_shared
== Compile MC module with call_shared; linkfile is PIC

ELD / out LLST / mobj $SYSTEM.SYSTEM.CCPLMAIN  &
  -libvol $myvol.svol -lib NDLL &
  -o MEXE

== Build MEXE, specifying CCPLMAIN (CRE component).
```

Compiling and Linking TNS/X Native C and C++ Programs

The TNS/X native C and C++ compilers take a module (a translation unit) as input and generate an object file or linkfile. A module is defined as a source file with all the headers and source files, except for any source lines skipped as the result of conditional preprocessor directives.

The `xld` utility is the TNS/X native linker that links PIC (Position-Independent Code) linkfiles to produce PIC loadfiles.

The SQL compiler processes executable files and generates code for embedded SQL statements.

The native `c89`, `c99`, and `c11` utilities control the TNS/X native C and C++ compilation system in the Open System Services (OSS) environment.

For information on how to compile and link programs in the OSS environment, see the `c89(1)`, `c99(1)`, or `c11(1)` reference page online or in the *Open System Services Shell and Utilities Reference Manual*.

For information about compiling and binding native C++ programs using NSDEE, see the online help for NSDEE.

Selecting a Development Platform

A development platform consists of the hardware system and software environment available to compile, link, and run a program.

You can develop OSS programs regardless of whether the OSS environment is available on the system. However, you cannot run and test OSS programs on a system without the OSS environment.

It is easier to develop a program in the environment in which it runs, but you can develop a program in one environment that runs in another environment, with a few restrictions. However, compile times are much faster in the PC environment.

Development Platform Capabilities (TNS/X Native C and C++ Programs) describes the capabilities of each development platform.

Table 52: Development Platform Capabilities (TNS/X Native C and C++ Programs)

Capability	System With Guardian Environment	System With Guardian and OSS Environments	NSDEE on Windows PC
Use Guardian development tools for Guardian programs?	Yes	Yes	Yes
Use Guardian development tools for OSS programs?	Yes	Yes	Yes
Use OSS development tools for Guardian programs?	No	Yes	Yes

Table Continued

Use OSS development tools for OSS programs?	No	Yes	Yes
Run Guardian programs?	Yes	Yes	Yes
Run OSS programs?	No	Yes	Yes
Compile programs with embedded SQL?	Yes	Yes	Yes
Use PC-based development tools for Guardian and OSS programs?	No	No	Yes

These restrictions apply to developing Guardian programs with OSS tools:

- You cannot use the `RUNNABLE` and `SEARCH` pragmas. You can direct the `c89` utility to bind implicitly after a compilation by using the `-c` flag. Unless `-c` is specified, `c89` invokes the linker. You can also specify library files to be searched using the `c89 -L` flag.
- You cannot use the `SSV` pragma. You can specify search directories using the `c89 -I` flag.

Specifying Header Files

The macros and functions in the TNS/X native C run-time library are declared in header files. Each header file contains declarations for a related set of library functions and macros, in addition to variables and types that complete that set. If you use a function in the library, you should include the header file in which it is declared. You should not declare the routine yourself because the declarations in the header files have provisions for several situations that can affect the form of a given declaration, including:

- Whether the routine is implemented internally as a function or a macro
- Whether the function is written in a language other than C
- Whether you are compiling for the small-memory or large-memory model
- Whether you are compiling for the 16-bit or 32-bit (wide) data model
- Whether you are compiling for the NonStop environment
- Whether you are compiling for TNS mode or native mode
- Whether you are compiling to use the Advanced Encryption Standard (AES) functions

In addition, the header file prototype declarations enable the compiler to check parameters and arguments for compatibility, ensuring that function calls provide the correct number and type of arguments.

A single set of TNS/X native C library header files supports all three environments (Guardian, OSS, and PC). The locations of the header files in the three environments is summarized in **Locations of Header Files**.

NOTE: A PC environment (target is not supported). TNS/R is not supported.

Table 53: Locations of Header Files

Environment	Location
Guardian	$SYSTEM.SYSTEM and $SYSTEM.ZTCPIP
OSS	`/usr/include` and its subdirectories
PC	Version 3.0 or later

To specify header files, use the `#include` preprocessor directive. For example, to include the STDIO header file, specify in your object file:

```
#include <stdio.h>
```

You might also need to specify header files for the TNS/X native Standard C++ Library and Tools.h++ library to use functions contained in those libraries. For examples of `#include` directives for these libraries, see **Using the Standard C++ Library** on page 88 and **Accessing Middleware Using HPE C and C++ for NonStop Systems** on page 104. For information about including DLLs at link time, see **Determining Which DLLs are Required** on page 414. For information about including compiler intrinsic functions mapped to Advanced Encryption Standard (AES) instructions, see **AES Intrinsic Functions** on page 400.

You can specify locations to search for header files:

- In the Guardian environment, use the `SSV` pragma to specify a search list of subvolumes for files specified in `#include` directives. For more details, see the pragma **SSV** on page 308.
- In the OSS environment, use the `-I` flag to the `c89` utility to specify a search list of directories for files specified in # `include` directives. For more details, see the `c89(1)` reference page either online or in the *Open System Services Shell and Utilities Reference Manual*.
- In the PC command line environment (using the cross compilers), use the `-I` flag to the `c89` utility to specify a search list of directories for files specified in `#include` directives. For more details, see the document *Using the Command-Line Cross Compilers on Windows*.

While header files are optional (but strongly recommended) for programs that contain Guardian or OSS modules exclusively, header files are required for mixed-module programs. If you do not compile using header files, `xld` cannot correctly resolve external references to Guardian and OSS versions of C functions.

AES Intrinsic Functions

The compiler intrinsic functions in TNS/X C/C++ are mapped to Advanced Encryption Standard (AES) instructions, which is an extension to the x86 instruction set architecture. The AES intrinsic functions operate on the `__m128i` type. To use intrinsic functions, include the prototypes from <builtin.h> header file. **Intrinsic Functions mapped to AES Instructions** lists the intrinsic functions mapped to AES instructions. For details about AES and Federal Information Processing Standards, see **http://csrc.nist.gov/publications/PubsFIPS.html**.

The AES intrinsic functions are supported from L15.08 RVU.

Table 54: Intrinsic Functions mapped to AES Instructions

AES Instruction	Description	Intrinsic Functions
AESENC	Perform one round of an AES encryption flow.	`__m128i _aesenc(__m128i x, __m128i y);`
AESENCLAST	Perform the last round of an AES encryption flow.	`__m128i _aesenclast(__m128i x, __m128i y);`
AESDEC	Perform one round of an AES decryption flow.	`__m128i _aesdec(__m128i x, __m128i y);`
AESDECLAST	Perform the last round of an AES decryption flow.	`__m128i _aesdeclast(__m128i x, __m128i y);`
AESKEYGENASSIST	Assist in AES round key generation. The int argument must be a compile time constant expression in the range of 0–255.	`__m128i _aeskeygenassist(__m128i x, const int c);`
AESIMC	Assist in AES Inverse Mix Columns.	`__m128i _aesimc(__m128i x);`
PCLMULQDQ	Carry-less multiply. The int argument must be a compile time constant expression in the range of 0–255.	`__m128i _pclmulqdq(__m128i x, __m128i y, const int i);`

SSE Intrinsic Functions

The compiler intrinsic functions in TNS/X C/C++ are mapped to Streaming SIMD Extensions (SSE) instructions, which is an extension to the x86 instruction set architecture. **Intrinsic functions mapped to packed data SSE instructions** lists the SSE instructions designed to help programming with AES intrinsics to manipulate packed data.

The SSE instructions are supported from L16.05 RVU.

Table 55: Intrinsic functions mapped to packed data SSE instructions

SSE instruction	Description	Intrinsic function
PADDB	Add packed byte integers.	`__m128i _paddb(__m128i a, __m128i b);`
PADDW	Add packed word integers.	`__m128i _paddw(__m128i a, __m128i b);`
PADDD	Add packed doubleword integers.	`__m128i _paddd(__m128i a, __m128i b);`

Table Continued

SSE instruction	Description	Intrinsic function
PADDQ	Add packed quadword integers.	`__m128i _paddq(__m128i a, __m128i b);`
PSUBB	Subtract packed byte integers.	`__m128i _psubb(__m128i a, __m128i b);`
PSUBW	Subtract packed word integers.	`__m128i _psubw(__m128i a, __m128i b);`
PSUBD	Subtract packed doubleword integers.	`__m128i _psubd(__m128i a, __m128i b);`
PSUBQ	Subtract packed quadword integers.	`__m128i _psubq(__m128i a, __m128i b);`
POR	Bitwise logical OR.	`__m128i _por(__m128i x, __m128i y);`
PXOR	Logical exclusive OR.	`__m128i _pxor(__m128i x, __m128i y);`
XORPS	Bitwise logical XOR for 4 packed single-precision floating-point values.	`__m128i _xorps(__m128i x, __m128i y);`
SHUFPS	Shuffle packed single-precision float-point values.	`__m128i _shufps(__m128i x, __m128i y, const int c);`
PSHUFB	Shuffle packed bytes.	`__m128i _pshufb(__m128i x, __m128i y);`
PSHUFD	Shuffle packed doubleword.	`__m128i _pshufd(__m128i x, const int c);`
PSLLDQ	Shift packed double quadword left logical.	`__m128i _pslldqi(__m128i x, const int c);`
PSRLDQ	Shift packed double quadword right logical.	`__m128i _psrldqi(__m128i x, const int c);`
PSLLW	Shift packed word values left logical.	`__m128i _psllwi(__m128i x, const int c);`
PSLLD	Shift packed double words left logical.	`__m128i _pslldi(__m128i x, const int c);`
PSRLD	Shift packed double words right logical.	`__m128i _psrldi(__m128i x, const int c);`

Table Continued

SSE instruction	Description	Intrinsic function
PSRLW	Shift packed word values right logical.	`__m128i _psrlwi(__m128i x, const int c);`
LOADDQU	Load 128 bits from unaligned memory address.	`__m128i _loaddqu_LE(const char _ptr64 * ptr);`
STOREDQU	Store 128 bits to unaligned memory address without byte swap.	`void _storedqu_LE(char _ptr64 * ptr, __m128i val);`
_mm_cvtsi128_si32	Copy the lower 32-bit integer from parameter to return value.	`__m128i _mm_cvtsi32_si128(int x);`
_mm_cvtsi128_si64	Copy the lower 64-bit integer from parameter to return value.	`__m128i _mm_cvtsi64_si128(int64 x);`
_mm_set_epi8	Set packed 8-bit integer return value with the supplied values.	`__m128i _mm_set_epi8(char v15, char v14, char v13, char v12,char v11, char v10, char v09, char v08,char v07, char v06, char v05, char v04,char v03, char v02, char v01, char v00);`
_mm_setr_epi8	Set packed 8-bit integer return value with the supplied values in reverse order.	`__m128i _mm_setr_epi8(char v00, char v01, char v02, char v03,char v04, char v05, char v06, char v07,char v08, char v09, char v10, char v11, char v12, char v13, char v14, char v15);`
_mm_set_epi32	Set packed 32-bit integer return value with the supplied values.	`__m128i _mm_set_epi32(int v3, int v2, int v1, int v0);`
_mm_cvtsi32_si128	Copy 32-bit integer from parameter to lower element of return value and zero upper element of return value.	`__m128i _mm_cvtsi32_si128(int x);`
_mm_cvtsi64_si128	Copy 64-bit integer from parameter to lower element of return value and zero upper element of return value.	`__m128i _mm_cvtsi64_si128(int64 x);`

Atomic Intrinsic Functions

Starting from L18.08, the following atomic intrinsic functions are supported:

Intrinsic Function	Description
`int8_t _atmAdd8(volatile int8_t _ptr64 *, int8_t);`	The 8-bit atomic add. The 16-bit, 32-bit, 64-bit variants of the function are supported from earlier RVUs.

`int8_t _atmSub8(volatile int8_t _ptr64 *, int8_t);`
`int16_t _atmSub16(volatile int16_t _ptr64 *, int16_t);`
`int32_t _atmSub32(volatile int32_t _ptr64 *, int32_t);`
`int64_t _atmSub64(volatile int64_t _ptr64 *, int64_t);`
	The 8-bit, 16-bit, 32-bit, and 64-bit atomic subtraction.

`int8_t _atmAnd8(volatile int8_t _ptr64 *, int8_t);`
`int16_t _atmAnd16(volatile int16_t _ptr64 *, int16_t);`
`int32_t _atmAnd32(volatile int32_t _ptr64 *, int32_t);`
`int64_t _atmAnd64(volatile int64_t _ptr64 *, int64_t);`
	The 8-bit, 16-bit, 32-bit, and 64-bit atomic bitwise AND.

`int8_t _atmXor8(volatile int8_t _ptr64 *, int8_t);`
`int16_t _atmXor16(volatile int16_t _ptr64 *, int16_t);`
`int32_t _atmXor32(volatile int32_t _ptr64 *, int32_t);`
`int64_t _atmXor64(volatile int64_t _ptr64 *, int64_t);`
	The 8-bit, 16-bit, 32-bit, and 64-bit atomic bitwise exclusive OR.

`int8_t _atmOr8(volatile int8_t _ptr64 *, int8_t);`
`int16_t _atmOr16(volatile int16_t _ptr64 *, int16_t);`
`int32_t _atmOr32(volatile int32_t _ptr64 *, int32_t);`
`int64_t _atmOr64(volatile int64_t _ptr64 *, int64_t);`
	The 8-bit, 16-bit, 32-bit, and 64-bit atomic bitwise OR.

Table Continued

`uint8_t _atmLoad8(volatile uint8_t _ptr64 *);`
`uint16_t _atmLoad16(volatile uint16_t _ptr64 *);`
`uint32_t _atmLoad32(volatile uint32_t _ptr64 *);`
`uint64_t _atmLoad64(volatile uint64_t _ptr64 *);`
	The 8-bit, 16-bit, 32-bit, and 64-bit atomic bitwise load.

`void _atmStore8(volatile uint8_t _ptr64 *, uint8_t);`
`void _atmStore16(volatile uint16_t _ptr64 *, uint16_t);`
`void _atmStore32(volatile uint32_t _ptr64 *, uint32_t);`
`void _atmStore64(volatile uint64_t _ptr64 *, uint64_t);`
	The 8-bit, 16-bit, 32-bit, and 64-bit atomic bitwise store.

`uint8_t _atmBoolCmpXch8 (volatile uint8_t _ptr64 *, uint8_t _ptr64 *old, uint8_t new);`
`uint8_t _atmBoolCmpXch16 (volatile uint16_t _ptr64 *, uint16_t _ptr64 *old, uint16_t new);`
`uint8_t _atmBoolCmpXch32 (volatile uint32_t _ptr64 *, uint32_t _ptr64 *old, uint32_t new);`
`uint8_t _atmBoolCmpXch64 (volatile uint64_t _ptr64 *, uint64_t _ptr64 *old, uint64_t new);`
	T
	he 8-bit, 16-bit, 32-bit, and 64-bit atomic compare and swap. If the current value of *ptr is *old*, then write *new* into *ptr.

Other intrinsic functions

The intrinsic functions for the following operations are supported from L16.05 RVU:

- Determine the CPU ID
- Read the timestamp counter
- Generate random numbers

Other intrinsic functions table lists the intrinsic functions.

Table 56: Other intrinsic functions

Instruction	Description	Intrinsic Functions

CPUID	Identifies the CPU. The 4 return registers EAX, EBX, ECX and EDX are stored in
	cpu_info[0]
	,
	cpu_info[1]
	,
	cpu_info[2]
	and
	cpu_info[3]
	.
		`void _cpuid(int cpu_info[4], int function_id);`
		`void _cpuidex(int cpu_info[4], int function_id, int subfunction_id);`

RDTSC	Read the timestamp counter.	`uint64 _rdtsc(void);`
RDRANDW	Read a word random number and store in the destination register.	`int _rdrand16(unsigned short* val);`
RDRANDL [1]	Read a doubleword random number and store in the destination register.	`int _rdrand32(unsigned int* val);`
RDRANDQ [1]	Read a quadword random number and store in the destination register.	`int _rdrand64(uint64* val);`
RDSEEDW	Read a word NIST SP800-90B & C complaint random value and store in the destination register.	`int _rdseed16(unsigned short* val);`
RDSEEDL [2]	Read a doubleword NIST SP800-90B & C complaint random value and store in the destination register.	`int _rdseed32(unsigned int* val);`
RDSEEDQ [2]	Read a quadword NIST SP800-90B & C complaint random value and store in the destination register.	`int _rdseed64(uint64* val);`

[1] Supported from Intel Ivy Bridge CPUs.

[2] Supported from Intel Broadwell CPUs.

Compiling and Linking Floating-Point Programs

You can now choose either Tandem floating-point format or IEEE floating-point format for performing floating-point arithmetic in your native C and C++ programs. This table compares the two formats.

Tandem Floating-Point Format	IEEE Floating-Point Format
	Default for TNS/X modules
A proprietary implementation of floating-point arithmetic that is supported in software millicode	An industry-standard data format that is supported in the processor hardware
Provides backward compatibility with pre-G06.06 C and C++ applications	Requires L15.02 for the TNS/X native C and C++ compilers.
Available for TNS C and C++, FORTRAN, TAL, pTAL, D-series Pascal, COBOL, and native C and C++ programs	Available only for native C and C++ programs. Used internally by TNS/X native COBOL programs
Requires conversion routines for data interchange between Tandem format and IEEE format (see the *Guardian Procedure Calls Reference Manual* for more detail about conversion routines)	Provides easier data interchange with other systems using 64-bit and 32-bit IEEE floating-point data formats; interchange typically consists of reversing the byte order to convert between big-endian data format (on NonStop systems) and little-endian format (on the target system)
	Provides better handling of exception conditions such as overflow and underflow; the existence of NaN (not a number), infinities, and exception flags make it easier to detect invalid results

To use IEEE floating-point format, you can optionally specify the IEEE_FLOAT pragma on the command line when running the TNS/X native C or C++ compiler. IEEE_FLOAT is the default setting for TNS/X native compilations.

If you are using TNS/X native C++, you also need to specify the `VERSION2`, `VERSION3`, or `VERSION4` directive. For more details, see the pragmas **IEEE_FLOAT** on page 248, **VERSION2** on page 324, **VERSION3** on page 326, and **VERSION4** on page 329.

The native compilers set the floating-point format type in the generated object file.

Two examples of compiling native C and C++ programs that use IEEE floating-point format:

```
> CCOMP / IN SOURCEA, OUT $.#LIST / OBJECTA; IEEE_FLOAT


> CPPCOMP / IN SOURCEB, OUT $S.#LIST / OBJECTB; VERSION2, &
  IEEE_FLOAT
```

Using Link Options to Specify Floating-Point Format

When linking object files using the `xld` utility, you can specify the floating-point state of the output object file using the `-set floattype` flag. You can set any of these options:

```
TANDEM_FLOAT
IEEE_FLOAT
NEUTRAL_FLOAT
```

If the -set floattype flag is not specified, the xld utility derives the floating-point state of the output object file from the states of the input files.

When modifying an existing object file, xld sets the state as specified by the –change floattype flag.

Link-Time Consistency Checking

The xld utility checks the consistency of floating-point type combinations when linking object files. The checking differs according to whether the –set flag is specified.

When the floattype attribute is not explicitly set with the –set flag, xld uses the floattype attribute values of all the input object files for determining the floattype attribute value for the output object file. If the consistency checks of the input object files result in an invalid floating-point state or inconsistent value, an error message is generated and no output object file is created.

Floating Point Consistency Check by xld Utility shows the results of each combination of floating-point states when the floattype attribute is not explicitly specified. The columns Tandem, IEEE, and Neutral, show the number of input object files characterized by that floating-point setting.

Table 57: Floating Point Consistency Check by xld Utility

Tandem	IEEE	Neutral	Consistency Check Result
1 or more	0	0 or more	No message is generated and the output object file has TANDEM_FLOAT state.
0	1 or more	0 or more	No message is generated and the output object file has IEEE_FLOAT state.
1 or more	1 or more	0 or more	Error message is generated. No output object file is created.
0	0	1 or more	No message is generated and the output object file has NEUTRAL_FLOAT state.

When the floattype attribute is explicitly specified with the -set flag, xld sets the floattype attribute value for the output object file to that specified value. If an inconsistency is detected, a warning message and an output object file are generated. If the floating-point state is invalid, no output object file is created.

Any floating type combination is allowed if the user explicitly overrides the default with the set floattype flag.

Floating-Point State as Determined by xld floattype Attribute shows the results of each floating-point state when the floattype attribute is explicitly specified.

Table 58: Floating-Point State as Determined by `xld floattype` Attribute

Tandem	IEEE	Neutral	-set Tandem	-set IEEE	-set Neutral
1 or more	0	0 or more	No message is generated and the output object file is `TANDEM_FLOAT` .	Warning message is generated and the output object file is `IEEE_FLOAT` .	Warning message is generated and the output object file is `NEUTRAL_FLOAT` .
0	1 or more	0 or more	Warning message is generated and output object file is `TANDEM_FLOAT` .	No message is generated and the output object file is `IEEE_FLOAT` .	Warning message is generated and the output object file is `NEUTRAL_FLOAT` .
1 or more	1 or more	0 or more	Warning message is generated and output object file is `TANDEM_FLOAT` .	Warning message is generated and the output object file is `IEEE_FLOAT` .	Warning message is generated and the output object file is `NEUTRAL_FLOAT` .
0	0	1 or more	Warning message is generated and output object file is `TANDEM_FLOAT` .	Warning message is generated and the output object file is `IEEE_FLOAT` .	No message is generated and the output object file is `NEUTRAL_FLOAT` .

Run-Time Consistency Checking

Run-time consistency checking includes a check for `floattype` consistency between the program file and the user library file, if one is used. IEEE and Tandem floating-point formats use different data formats and calling conventions: IEEE floating-point values are typically passed in IEEE floating-point registers; while Tandem floating-point values are passed in general-purpose registers. Therefore, problems can occur if a function using one floating-point format calls a function using the other format. Checks are performed at process creation to ensure that the user library (if there is one) is marked with a `floattype` consistent with the program file.

These combinations are considered to be conflicting and are not allowed to begin to run:

`floattype` in program file	`floattype` in user library file
IEEE	Tandem
Tandem	IEEE
Neutral	IEEE

The case in which the floattype attribute of the program file is IEEE and the user library file is Neutral is not considered a conflict. In this case, the program is allowed to execute, and the C/C++ run-time libraries operate in IEEE mode.

If you are using a native user library, the library file should use the same floating-point format as the program, and the library should be marked accordingly. If the user library doesn't use floating point at all, you can mark the library NEUTRAL_FLOAT using the xld -set floattype or -change floattype command. Then the user library can be used by any type of program.

Even if the user library is marked with a floattype attribute that conflicts with the program file, the program can use the library if it does not call anything in the user library that uses floating point. In this case, you need to mark the program file with the xld -set float_lib_overrule on command to disregard the floattype attribute of the user library file.

In fact, the run-time consistency check can be overruled by using the -set float_lib_overrule on flag of the xld utility. If you overrule the consistency check, the operating system allows a floating-point inconsistency between the user library and the program. If you do not set float_lib_overrule, and there is an inconsistency between the program file and user library, the operating system generates an error code and does not run the program.

Linking Mixed-Language Programs

When linking mixed-language programs that use IEEE floating-point format, specify the -set floattype IEEE flag using the xld utility.

For example, if you have a mixed-language program composed of a C module that uses IEEE floating-point format, and a COBOL module that doesn't use floating point but is marked by the native COBOL compiler as TANDEM_FLOAT, then you could use the xld -set floattype IEEE_FLOAT command. Or you could first use the xld -change command to change the COBOL object file to NEUTRAL_FLOAT.

This example illustrates linking a mixed-language program that uses IEEE floating-point format:

```
> XLD $SYSTEM.SYSTEM.CCPMAINX COBJ XPTALOBJ &
  -set floattype ieee_float -o MYEXEC
```

In this example, the native C object file named COBJ uses IEEE floating-point format, and the XPTAL-created object file named XPTALOBJ uses Tandem floating-point format. (Note that XPTAL supports only Tandem floating-point format.) To link these modules, you must specify the -set floattype IEEE_FLOAT flag. If this flag is not specified, xld generates error messages because of the mismatch between Tandem and IEEE floating-point formats. When this flag is specified, xld generates a warning message about the mismatch and builds the executable file MYEXEC. CCPMAINX (an object file) is a standard item, required when linking C programs.

Use the -set and -change flags of the xld utility to set or change the float_lib_overrule attribute when creating object files. If the float_lib_overrule is specified more than once by either the -set or -change flags, all occurrences except the last one are ignored. The float_lib_overrule attribute can be changed only for executable files. An error occurs if an attempt is made to change the value of this attribute for relinkable files.

For More Information

See these manual for more details about using floating-point formats:

- *Guardian Programmer's Guide* for information about building programs using IEEE floating-point format
- *Guardian Procedure Calls Reference Manual* for information about operating mode routines and routines for converting between Tandem and IEEE floating-point formats
- The descriptions of pragmas **IEEE_FLOAT** on page 248 and **TANDEM_FLOAT** on page 319.

Working in the Guardian Environment

In the Guardian environment, you can compile and link programs for either the Guardian or Open System Services (OSS) environment.

Compiling a Module

The native compilers translate the source text of a module and produce:

- An extensive compiler listing; several pragmas enable you to control the content of this compiler listing
- A nonexecutable object file (if the compiler encountered no errors during the compilation)

After compiling all the modules that compose your program, you collect and combine them into a program file (an executable object file) by using the `xld` utility.

If your program comprises a single module, you can use the `RUNNABLE` pragma to direct the compiler to produce a program file instead of a nonexecutable object file.

If your program comprises more than one module, you can use the `RUNNABLE` and `LINKFILE` pragmas to direct the compiler to produce a linked program file instead of a nonexecutable object file. For more details, see pragma **LINKFILE** on page 262.

To use the `RUNNABLE` pragma, one of the modules must contain the main function of the program.

The CCOMP command invokes the TNS/X native C compiler. The CPPCOMP command invokes the TNS/X native C++ compiler. The syntax for these commands is shown in this diagram.

```
[ RUN ] { CCOMP | CPPCOMP } / IN source [ , OUT listing ]
      [ , run-options ] / [ object ]
      [ ; compile-option [ , compile-option ]... ]

compile-option:
      { pragma                                          }
      { define identifier [ constant ] }
      { undefine identifier                   }
```

[RUN] CCOMP

is the TACL command to start the TNS/X native C compiler process. The RUN command keyword is optional.

[RUN] CPPCOMP

is the TACL command to start the TNS/X native C++ compiler process. The RUN command keyword is optional.

IN *source*

specifies the primary source file of the module. The file must be a valid Guardian file name for either a type 101 (EDIT) or type 180 disk file. Interactive input from a terminal or a process is not accepted.

OUT *listing*

specifies the file to which the native C compiler writes the compiler listing. When specified, *listing* is usually a spooler location. If you omit the OUT option, the compiler writes the listing to your current default output file. If the file already exists, the compiler attempts to delete the file and continue.

run-options

is a comma-separated list of additional options for the RUN command. These options are described in the *TACL Reference Manual*.

object

specifies the file to which the TNS/X native C or C++ compiler writes the object code for the source text. If you do not specify an object file, the compiler writes the object code to the file OBJECT in your current default volume and subvolume. If OBJECT cannot be created, the compiler writes the object code to the file ZZLDF*nnn* (where *nnn* is a unique three-digit number) in your current default volume and subvolume.

compile-option

modifies compiler operation by specifying a compiler pragma or defining a preprocessor symbol.

pragma

is any valid command-line compiler pragma. The COLUMNS, ENV, FIELDALIGN, FUNCTION, INNERLIST, LIST, MAP, OVERFLOW_TRAPS, POP, PUSH, and WARN pragmas are supported.

define *identifier [constant]*

defines *identifier* as a preprocessor symbol. If *identifier* is followed by a constant, *identifier* is defined as an object-like macro that expands to the given value. `define` is equivalent to using the `#define` preprocessor directive in source text.

undefine *identifier*

deletes *identifier* as a preprocessor symbol. Using undefine is equivalent to using the `#undef` preprocessor directive in source text.

Usage Guidelines

- The compiler accesses source files as text-type logical files. Consequently, the source files you specify in a module must represent physical file types that the compiler can access as text-type logical files.
- The compiler returns one of these completion codes:

0	The compilation completed successfully.
1	The compilation completed with warnings (but no errors).
2	The compilation completed with errors.
3	The compiler terminated abnormally as the result of an internal error.

Examples

1. Directing the compiler to translate the source file ABSC (which contains an entire C++ program), sending the compiler listing to the device $S.#ABSL, storing the object file under the name ABSO, and using the default VERSION3 library:

   ```
   > CPPCOMP /IN absc, OUT $s.#absl/ abso;
   ```

2. Disabling the run-time diagnostics that the `assert` function can issue by defining `NDEBUG` as a preprocessor symbol:

   ```
   > CCOMP /IN appc, OUT $s.#appl/ appo; define NDEBUG
   ```

3. Generating an object file for the OSS environment:

   ```
   > CCOMP /IN filec / fileo; SYSTYPE OSS
   ```

4. Generating an executable program from a single module:

   ```
   > CCOMP /IN filec / fileo; RUNNABLE
   ```

5. Generating an executable program composed of the modules `mod1c`, `mod2o`, `mod3o`, and `mod4o`:

   ```
   > CCOMP /IN mod1c / fileo; RUNNABLE, LINKFILE "myfile"
   ```

   In this example, `myfile` is a text file containing these names of object files (not source files):

   ```
   mod2o
   mod3o
   mod4o
   ```

Linking a TNS/X Module

You need to use a linker to collect and combine object files into a loadfile (which means, either an executable program or a DLL) if you did not instruct the compiler to perform the linking (that is, if you did not use the `RUNNABLE` pragma when compiling a single-module program or if you did not use the `RUNNABLE` and `LINKFILE` pragmas when compiling a multiple-module program).

This subsection provides a summary of linking TNS/X native C and C++ programs using the `xld` utility, the TNS/X native linker. The `xld` utility is a linker for PIC (Position-Independent Code). For more details about PIC and sharing code, see:

- Pragma **CALL_SHARED** on page 210
- Pragma **SHARED** on page 300
- Example of compiling and linking PIC using `xld` in **Examples** on page 381

CCPMAINX File

The CCPMAINX file, located in $SYSTEM.SYSTEM, contains initialization code for the TNS/X native C and C++ run-time libraries. This object file must be linked into C and C++ programs. In many instances, `c89`, CCOMP and CPPCOMP link, compile, and automatically include CCPMAINX.

You need to explicitly link CCPMAINX when you run the linker to create a loadfile if the linkfiles were:

- Compiled without the `RUNNABLE` and `LINKFILE` options when using CCOMP or CPPCOMP
- Compiled with the `-c`, `-Wnolink`, or `-Wnomain` option when using `c89`

A similar initialization file for doing fault-tolerant programming, named CRTLNSX, also might need to be linked. On OSS environment, the file is named `crtlnsx.o`.

NOTE: For programs compiled with the LP64 data model, the name of the main program file is CMAIN64X.

Dynamic-Link Libraries (DLLs)

A DLL contains code present in virtual memory at run time, to be shared by other processes, rather than code that is linked into object files. A DLL can also contain global data, and each process using the DLL automatically gets its own run-time copy of the data, called instance data. A process can use several DLLs.

HPE supplies public DLLs; you can create your own DLLs.

Each process loads its public DLLs at startup from the active ZDLL*nnn* subvolume. The ZDLL*nnn* subvolume, located on $SYSTEM, contains a version of the public DLLs supplied by HPE. A node can have more than one ZDLL*nnn* subvolume, but only one active (running) ZDLL*nnn* subvolume.

When a program or DLL is linked and it makes references to symbols not defined within that same program or DLL, those symbols might come from other DLLs. So, at the time of linking, the linker looks for those symbols in other DLLs, and the user must give `-l` options to the linker to inform it about the other DLLs. However, the linker also knows how to look for symbols in the operating system kernel without the need for any `-l` options. In fact, the kernel itself is a set of DLLs. Since the symbols in the kernel are found by the linker without the need for any explicit `-l` options, the DLLs that constitute the kernel are called the “implicit” DLLs.

When you use `xld` to build an executable program, `xld` fixes up references to the DLLs that you have specified. You can also use `xld` to repeat the fix-up process on an existing program to use a new version of an DLL or let the `rld` run-time loader or operating system update the references when you execute the program.

To create an executable TNS/X native C or C++ program using `xld`:

1. Specify $SYSTEM.SYSTEM.CCPMAINX to link to the TNS/X native C and C++ run-time library initialization object code file.
2. Specify the object code files that you compiled.
3. Specify the DLLs used by your program. For complete details, see **Determining Which SRLs are Required** on page 378.
4. If your program uses the active backup programming functions, such as `__ns_start_backup()`, specify the active backup programming support object code file $SYSTEM.SYSTEM.CRTLNSX.

Command examples are shown in **Examples** on page 381.

Shared Run-Time Libraries and Dynamic-Link Libraries (DLLs)

The NonStop system implicit libraries are compatible with the use of dynamic-link libraries (DLLs). The system implicit DLLs are public libraries in PIC (Position-Independent Code) format.

You use the `xld` utility to create a linkable object file or linkfile. You can also use `xld` to create an executable (known as a loadable object file or a loadfile) in PIC format that can function as a dynamic-link library (DLL).

Shared run-time libraries are not supported for TNS/X code.

Determining Which DLLs are Required

The DLLs that are required by a program depend on whether the program:

- Runs in the NonStop environment
- Uses the C run-time library
- Uses the C++ run-time library
- Uses the Tools.h++ library (and whether Version 6.1 or Version 7)
- The Standard C++ Library
- The TCP/IP sockets library

Below table is a conceptual stack of the DLLs that make up the context of VERSION2, VERSION3, and VERSION4 . In this table, CPPC represents the DLL named XCPPCDLL or WCPPCDLL (C run-time library for 32-bit and 64-bit variants). XCPP2DLL is given as CPP2 (the VERSION2 Standard C++ Library); similarly, CPP3 and CPP4 are used for Version3 and Version4 respectively.

For a given version, you can link only to the DLLs listed in the column or stack for that version in the below table . If you attempt to link to DLLs that are listed in another column, unexpected results can occur. The order of the column or stack is also important, especially for TNS/X native C++, because it represents the order in which libraries should be loaded and accessed at run time.

For example, if you are using VERSION2 on Guardian, you can link to XTLH7DLL (Tools.h++ version 6), but you cannot link to XCPP3DLL or WCPP3DLL. Similarly, if you are using VERSION3, you cannot link to XCPP2DLL (the VERSION2 Standard C++ Library).

Table 59: DLLs Available When Using VERSION2 and VERSION3 and VERSION4

Version 2 Guardian	OSS	Version 3 Guardian	OSS	Version 4 Guardian	OSS
TLH7	TLH7	N.A.	N.A.	N.A.	N.A.
CPP2 CPPC CRTL CRE	CPP2 CPPC CRTL CRE	CPP3 CPPC CRTL CRE	CPP3 CPPC CRTL CRE	CPP4 CPPC CRTL CRE	CPP4 CPPC CRTL CRE
OS	OSS	OS	OSS	OS	OSS
OS	OS	OS	OS	OS	OS

Notes: DLL names are shown without the leading Z and these DLL; therefore, CPPC represents XCPPCDLL or WCPPCDLL, and TLH7 represents XTLH7DLL or WTLH7DLL.

OS represents the HPE NonStop OS.

OSS represents the Open System Services environment.

CRTL represents the C run-time library. CRE represents the common run-time environment. For other environments such as the PC, the names are prefixed with lib and suffixed with .so; therefore, CPPC is libcppc.so.

For more details, see:

- **Using the Standard C++ Library**
- Pragmas **VERSION2**, **VERSION3**, and **VERSION4**
- **C++ Run-Time Library and Standard C++ Library** on page 29

Additional DLLs might be required for other run-time libraries and services. For example, if you use the `VERSION2` pragma with the CPPCOMP driver in the Guardian environment (or the `-Wcplusplus` and `-Wversion2` flags with the `c89` driver in the OSS environment), these DLLs are automatically searched:

- XCPPCDLL or WCPPCDLL (the common C++ run-time DLL for 32-bit and 64-variants)
- XCPP2DLL (the `VERSION2` Standard C++ library DLL)

If, however, you invoke `xld` explicitly, you need to include these DLLs in the link command using the `-lib` option, as shown in the following table.

Table 60: Using the Guardian xld Utility to Link DLLs

If your program uses:	You should specify these xld utility flags:
C run-time library and runs in the Guardian environment	`-l CRTL -l CRED`
C run-time library and runs in the OSS environment	`-l CRTL -l CRE -l OSSK -l OSSF -l SEC -l I18N -l ICNV -l OSSE -l INET -l OSSH`
`VERSION2` C++ run-time library and runs in the Guardian environment	`-l CPP2 -l CPPC -l CRTL -l CRE`
`VERSION2` C++ run-time library and runs in the OSS environment	`-l CPPC -l CPP2 -l CRTL -l CRE -l OSSK -l OSSF -l SEC -l I18N -l ICNV -l OSSE -l INET -l OSSH`
`VERSION2` Standard C++ Library, Tools.h++ (version 7) and runs in the Guardian environment	`-l TLH7 -l CPP2 -l CPPC -l CRTL -l CRE -l SEC -l I18N -l ICNV -l INET`
`VERSION2` Standard C++ Library, Tools.h++ (version 7) and runs in the OSS environment	`-lCPPC -1 CPP2 -1 CRE -1 CRTL -1 OSSK -1 OSSF -1 SEC -1 I18N -1 ICNV -1 OSSE -1 INET -1 OSSH -1 OSSC`
`VERSION3` Standard C++ Library and runs in the OSS or Guardian environment	`-lCPPC -1 CPP3 -1 CRE -1 CRTL -1 OSSK -1 OSSF -1 SEC -1 I18N -1 ICNV -1 OSSE -1 INET -1 OSSH -1 OSSC`

Table Continued

If your program uses:	You should specify these xld utility flags:
`VERSION4` Standard C++ Library and runs in the OSS or Guardian environment	`-lCPPC -1 CPP4 -1 CRE -1 CRTL -1 OSSK -1 OSSF -1 SEC -1 I18N -1 ICNV -1 OSSE -1 INET -1 OSSH -1 OSSC`
OSS `nlist()` function	`-l UTIL` and other DLLs required by the program environment
TCP/IP socket library	`-l INET` and other DLLs required by the program environment

Examples

1. The specified `xld` flags link a `VERSION2` Guardian C++ program that uses the Tools.h++ class libraries (TLH7) and the Standard C++ Library (CPP2):

   ```
   > XLD $SYSTEM.SYSTEM.CCPMAINX MYOBJ -o MYEXEC   &
         -l TLH7  -l CPPC  -l CPP2 -l CRTL &
         -l CRE
   ```

2. The specified `xld` flags link a `VERSION2` OSS C program:

   ```
   > XLD $SYSTEM.SYSTEM.CCPMAINX MYOBJ -o MYEXEC &
        -set systype oss &
        -l TLH7 -l CPPC -l CPP2 &
        -l OSSH -l CRTL -l CRE &
        -l OSSK -l OSSF -l SEC &
        -l I18N -l ICNV -l OSSE &
        -l INET
   ```

3. Linking with the `xld` linker and the `VERSION3` Standard C++ Library (the default library):

   ```
   > XLD $SYSTEM.SYSTEM.CCPMAINX MYOBJ -o MYEXEC &
        -l CPP3 -l CPPC
   ```

4. The specified `xld` flags link a LP64 OSS C program:

   ```
   > XLD $SYSTEM.SYSTEM.CMAIN64X MYOBJ -o MYEXEC &
             -set systype oss &
             -l CRTl -l CRE -l OSSK -l OSSF -l SEC &
             -l I18N -l ICNV -l OSSE -l INET -l OSSH
   ```

5. Compiling PIC (Position-Independent Code) using the default TNS/X native C++ dialect (`VERSION3`). The example program (MEXE) uses a DLL (named NDLL) compiled from a library file named NC, which contains the `getnum()` function. MEXE imports `getnum()` and prints the result (31).

   Note the use of the `import$` and `export$` keywords, the `SHARED` pragma (to compile the library) and `CALL_SHARED` pragma (to compile the main module), and the `xld` and `rld` utilities (the PIC linker and loader). The result is a dynamic-link library (DLL) named NDLL:

   `Source file (named MC):`

   ```
   import$ extern int getnum();

   int main()
   {
     int x = -99;
     x = getnum();
     printf (“x was -99; is now %d”, x);
     return 0;
   ```

```
}
Library for DLL (file name NC):

export$ int getnum()
{
  return 31;
}
```

Compiler and Linker Commands:

```
CCOMP / in NC, out NLST / NDLL; shared
== Compile NC with shared (as a DLL); linkfile is PIC

CCOMP / in MC, out MLST / MOBJ; call_shared
== Compile MC module with call_shared; linkfile is PIC

XLD / out LLST / mobj $SYSTEM.SYSTEM.CCPMAINX  &
  -libvol $myvol.svol -lib NDLL &
  -o MEXE

== Build MEXE, specifying CCPMAINX (CRE component).
```

Using the Native C/C++ Cross Compiler on the PC

NSDEE

The HPE NonStop Development Environment for Eclipse (NSDEE) is a Windows hosted integrated development environment for NonStop applications. NSDEE supports building NonStop applications locally using Windows-hosted cross compilers and tools, or remotely using compilers and tools on a NonStop server. NSDEE also provides facilities for transferring source files and binaries to and from NonStop systems as well as facilities for editing remote source files locally.

NSDEE Core with Debugging also includes a separately installed integrated debugger that you can use to debug TNS/E and TNS/X processes.

For more information about NSDEE, including RVU support, cross compiler support, software requirements, and hardware requirements, see the documentation for NSDEE that is integrated with the software. After you install the software, you can access the documentation from the Eclipse Workbench help menu:

Procedure

1. To access the Workbench help menu, select **Help** > **Help Contents**.
2. From the list of manuals, select the *NonStop Development Environment for Eclipse User Guide*.
3. If you have installed NSDEE Core with Debugging, you can also select the *NonStop Development Environment for Eclipse Debugging Supplement* from the list of manuals.

ETK

ETK is a GUI-based extension package to Visual Studio.NET that provides full application development functions targeted for HPE NonStop servers. Developing, editing, and building are very similar between Visual Studio.NET and ETK.

ETK is an independent product delivered on a separate compact disc and is not available on the HPE NonStop Site Update Tape (SUT). You must install Visual Studio.net on your PC before installing ETK. Several cross compilers (C/C++, COBOL85, EpTAL, and pTAL) and linkers (`eld`, `nld,` and `ld`) are available separately from ETK.

NOTE: ETK is not available for TNS/X systems.

Capabilities of ETK

ETK works with Visual Studio.NET, and the optional cross compilers to allow application developers to design, build, and deploy applications targeted at the NonStop server environment. ETK release 3 contains features that support TNS/E systems.

Using ETK, you can:

- Build native mode C/C++, COBOL, and pTAL applications targeted at NonStop systems
- Complete development functions more quickly by using a familiar user interface

- Create projects using wizards, including projects that contain embedded NonStop SQL/MP or NonStop SQL/MX source code
- Build Guardian and OSS executable (either PIC or G-series non-PIC), dynamic link libraries (DLLs), static libraries, or user libraries
- Compile, link, build, and deploy Guardian and OSS programs to create objects that execute only in the Guardian and OSS environments of NonStop systems
- Set and maintain NonStop compiler, linker, and tool options on a per-target and per-file basis
- Display and act on messages from ETK compilation tools in the same way as those provided by the Microsoft tools
- When multiple release versions are installed, choose any of these installed versions of the cross compilers, tools, and libraries
- Choose CodeWright as the default editor
- Deploy targets to a NonStop host automatically with project deployment or manually with the Transfer Tool
- Create and maintain archive files
- Modify the name of include files from the Guardian format to the PC format using the Fix Include Tool
- Compile NonStop SQL/MP statements embedded in native C and COBOL source code
- Compile NonStop SQL/MX statements embedded in native C/C++ and COBOL source code
- Enter ADD, MODIFY, SET, and DELETE statements into a TACL DEFINE file
- Configure external tools, such as Visual Inspect, to launch from the integrated development environment
- Integrate with source control tool
- Access online documentation for ETK with the Visual Studio.NET documentation, or viewed singly as a preferred collection.

Hardware and Software Requirements

ETK is supported on the Windows NT, Windows 2000, and Windows XP operating systems.

For the latest PC and NonStop server hardware and software requirements, review ETK online help.

Online Help

Online help is the only user documentation for ETK. The online help is composed of these components:

- Context-sensitive help for GUI objects
- Help topics, such as “Setting Compiler Options”
- Glossary of terms
- Tutorial introducing the application
- An HTML help file named “Using Command-Line Cross Compilers on Windows”

Usage Guidelines

The C/C++ cross compiler produces object code that runs on TNS/R, TNS/E, or TNS/X systems in either the Guardian or the OSS environment. The H-series and J-series cross compilers package has the ability to produce TNS/R (G-series), TNS/E (H-series and J-series), or TNS/X (L-series) object files.

Source code that is compilable by the native C and C++ compilers running on TNS/R, TNS/E, or TNS/X systems is also compilable by the native PC cross compiler, except for these differences:

- The default target platform for the PC cross compiler depends on the copy of the compiler used. For compilers installed in G-series RVU folders, the default is TNS/R servers. This compiler will not accept the `–Wtarget` option.

  For compilers installed in H-series RVU folders, the default is TNS/E servers (`–Wtarget=tns/e`). For compilers installed in L-series RVU folders, the default is TNS/X servers (`–Wtarget=tns/x`). To produce programs that run on TNS/R servers, specify the `-Wtarget=tns/r` flag to the `c89` utility.

- The default target environment for the PC cross compiler is the OSS environment. To produce programs that run in the Guardian environment, specify the `-Wsystype=guardian` flag of the `c89` utility.
- The C PC cross compiler supports embedded SQL. When you enable SQL, HPE C passes the SQL options you choose to `cfe` so that embedded SQL statements are compilable. Additional options are required on the PC as well.
- To use the C/C++ cross compiler, the source files must reside in the PC namespace. Because the cross compiler runs on the PC, the compiler cannot see files located on NonStop server nodes. Therefore, you must transfer your source files from the server to the PC using file transfer protocol (FTP) or SFTP.

  When your source files are on the PC, they can be distributed anywhere in the PC namespace. Likewise, any files that are output from the PC cross compilers can be distributed anywhere in the PC namespace.

- PC source files with `#include` path names use the backslash (\) separator. These path names are correctly interpreted by the PC cross compilers. However, PC path names cannot be used by the native compilers that run on NonStop servers.
- The PC cross compiler interprets the slash character (/) in `#include` path names as a backslash (\). Therefore, OSS source files with directory names can map automatically to the PC namespace.
- The PC cross compiler handles source-file name suffixes in the same manner as `c89` and `c99`. Source-file names must be identified with the *.suffix* format just as are OSS file names.
- Many products have shared run-time libraries (SRLs) or dynamic-link libraries (DLLs) for linking on the PC. If a product does not have an SRL or DLL on the PC, perform final linking on a NonStop server that does have the appropriate library file.

Cross Compilers

Several cross compilers work with NSDEE or ETK:

- **Introducing the CRE** on page 113
- **pTAL Cross Compiler** on page 422
- **EpTAL Cross Compiler** on page 423
- **XpTAL Cross Compiler** on page 423

- **COBOL85 Cross Compilers** on page 424
- **ECOBOL Cross Compiler** on page 424
- **XCOBOL Cross Compiler** on page 425

ETK cross compiler package is installed with an HTML help file (“Using the Command-Line Cross Compilers on Windows”).

C/C++ PC Cross Compiler

Using NSDEE or ETK, you can compile with the C/C++ PC cross compiler, which is essentially the same as the native C/C++ compiler available on HPE NonStop servers. The NSDEE and ETK PC cross compilers produce TNS/R native object code that runs on TNS/R servers or TNS/E native object code that runs on TNS/E servers. On TNS/X servers, NSDEE cross compilers produce TNS/X native object code.

NOTE: ETK is not supported on TNS/X systems.

You can invoke the cross compiler from the command line (DOS prompt):

- If you are using the C/C++ cross compiler named `c89`, version G06.14 and later.
- If you are using the C/C++ cross compiler named `c99`, version H06.21 and later.
- If you are using the C/C++ cross compiler named `c11`, the compiler supports C from version L16.02 and later, and both C/C++ from version L17.02 and later.

The command-line input format is similar to that of `c89` on the OSS platform.

You can embed SQL/MP or SQL/MX statements into native mode C/C++ source code from the command-line. You need to specify the host, user logon, and NonStop server location (Guardian subvolume or OSS directory). For more details, see:

- The online help for NSDEE or ETK
- The HTML help file for ETK named “Using Command-Line Cross Compilers on Windows”
- *SQL/MP Programming Manual for C*
- *SQL/MX Progamming Manual for C and COBOL*

pTAL Cross Compiler

The optional (for ETK) pTAL cross compiler compiles pTAL code that runs on TNS/R NonStop servers. The pTAL cross compiler:

- Targets only the Guardian environment, not OSS
- Can produce:
  - Object files suitable for input to the TNS/R native linker
  - Executable files suitable for running on TNS/R systems

    These files are identical, except for embedded file names, to objects and executable files created on other platforms by the pTAL compiler.

- Accepts compiler and linker options that you select through the Project command on the Options menu
- Does not support embedded SQL statements
- Can produce dynamic-link libraries (DLLs), static libraries, or user libraries

For more details about pTAL, see the *pTAL Programmer's Guide*.

EpTAL Cross Compiler

The optional (for ETK) EpTAL cross compiler compiles EpTAL code that will run on TNS/E NonStop servers. The EpTAL cross compiler:

- Targets only the Guardian environment, not OSS
- Can produce either:
    - Object files suitable for input to the `eld` linker
    - Executable files suitable for running on TNS/E systems

      These files are identical, except for embedded file names, to objects and executable files created on other platforms by the EpTAL compiler.
- Accepts compiler and linker options that you select through the Project command on the Options menu
- Does not support embedded SQL statements
- Can produce dynamic-link libraries (DLLs), static libraries, or user libraries

For more details about EpTAL, see the *pTAL Programmer's Guide*.

XpTAL Cross Compiler

The XpTAL cross compiler compiles XpTAL code that will run on TNS/X NonStop servers. The XpTAL cross compiler:

- Targets only the Guardian environment, not OSS
- Can produce either:
    - Object files suitable for input to the `xld` linker
    - Executable files suitable for running on TNS/X systems

      These files are identical, except for embedded file names, to objects and executable files created on other platforms by the XpTAL compiler.
- Accepts compiler and linker options that you select through the Project command on the Options menu
- Does not support embedded SQL statements
- Can produce dynamic-link libraries (DLLs), static libraries, or user libraries

For more details about XpTAL, see the *pTAL Programmer's Guide*.

COBOL85 Cross Compilers

The native COBOL85 cross compilers are an option available with ETK, and separately as command-line cross compilers for the G06.14 and later RVUs.

The native COBOL85 cross compilers allow you to:

- Write, compile, and link NonStop server applications (Guardian and Open System Services [OSS] executables, static libraries, and user libraries) on the PC and transfer them to the OSS or Guardian platform for use in production

  Object files built on the PC platform using the native COBOL85 compiler are compatible with object files built on the NonStop server platform using the NMCOBOL or ECOBOL compiler.

  The default target for the COBOL85 cross compilers is Guardian.

- Link NMCOBOL, C/C++, and pTAL objects into a single executable file
- When multiple RVUs are installed, choose any installed RVUs of the cross compilers, tools, and libraries
- On ETK platforms, enter ADD, MODIFY, SET, and DELETE statements into a TACL DEFINE file
- On ETK and command-line platforms, compile SQL/MP or SQL/MX statements embedded in native COBOL source code

  Your PC must be connected to the NonStop host for certain SQL compile-time operations and for running your applications.

The native COBOL85 cross compilers are delivered on a separate independent product CD and by means of Scout for NonStop Servers, and are not available on the SUT.

For more detail about the native COBOL85 cross compilers, see:

- The cross compiler documentation on the PC
- The *COBOL for TNS and TNS/R Manual*

ECOBOL Cross Compiler

The ECOBOL compiler is available as command-line cross compilers for the H06.08 and later RVUs and J06.03 and later RVUs.

The ECOBOL compiler allows you to:

- Write, compile, and link NonStop server applications (Guardian and Open System Services [OSS] executables, static libraries, and user libraries) on Windows and transfer them to the OSS or Guardian platform for use in production

  Object files built on the Windows platform using the ECOBOL compiler are compatible with object files built on the NonStop server platform using the ECOBOL compiler.

  The default target for the ECOBOL compilers is Guardian.

- Link ECOBOL, C/C++, and EpTAL objects into a single executable file
- When multiple RVUs are installed, choose any installed RVUs of the cross compilers, tools, and libraries

- On ETK platforms, enter ADD, MODIFY, SET, and DELETE statements into a TACL DEFINE file
- On ETK and command-line platforms, compile SQL/MP or SQL/MX statements embedded in native COBOL source code

  Your PC must be connected to the NonStop host for certain SQL compile-time operations and for running your applications.

The ECOBOL cross compiler is delivered on a separate independent product CD and by means of Scout for NonStop Servers, and are not available on the SUT.

For more details about the ECOBOL compiler, see:

- The cross compiler documentation on the PC
- The *COBOL for TNS/E and TNS/X Manual*

XCOBOL Cross Compiler

The XCOBOL compiler is available as command-line cross compilers for the L15.02 and later RVUs.

The XCOBOL compiler allows you to:

- Write, compile, and link NonStop server applications (Guardian and Open System Services [OSS] executables, static libraries, and user libraries) on Windows and transfer them to the OSS or Guardian platform for use in production

  Object files built on the Windows platform using the XCOBOL compiler are compatible with object files built on the NonStop server platform using the XCOBOL compiler.

  The default target for the XCOBOL compilers is Guardian.
- Link XCOBOL, C/C++, and XpTAL objects into a single executable file
- When multiple RVUs are installed, choose any installed RVUs of the cross compilers, tools, and libraries

The native XCOBOL cross compilers are delivered on a separate independent product CD and by means of Scout for NonStop Servers, and are not available on the SUT.

For more detail about the native XCOBOL compiler, see:

- The cross compiler documentation on the PC
- The *COBOL for TNS/E and TNS/X Manual*

PC Tools

Several PC tools work with ETK:

- **Visual Inspect** on page 426
- **ar Tool (File Archive)** on page 426

NOTE:

For information about the tools NSDEE supports, see the online help for NSDEE.

The Standard C++ Library Version 3 is supported by ETK. In addition, the Rogue Wave libraries are available with ETK. The Rogue Wave libraries include the Standard C++ Library Version 1 and Version 2 and Tools.h++ (both Versions 6.1 and 7). Both SQL/MP and SQL/MX are supported by ETK.

Visual Inspect

Visual Inspect is an optional PC-based (GUI) symbolic debugger designed to work in complex distributed environments. Visual Inspect:

- Supports application debugging in either the development or production environment
- Uses program visualization, direct manipulation, and other techniques to improve productivity for both new and sophisticated users
- Provides source-level debugging for servers executing in either the NonStop environment; provides additional application navigation features that allow a higher level of abstraction
- Supports TNS and native machine architectures and compilers (that is, C, C++, COBOL, TAL, pTAL, D-series Pascal, and FORTRAN), in both the Guardian and OSS environments.
- Is available also as a stand-alone product

For more details about enabling and using Visual Inspect, see the online help for Visual Inspect. Visual Inspect is not supported on TNS/X systems.

ar Tool (File Archive)

Use the `ar` tool to create and maintain file archives. The `ar` tool:

- Builds a file archive from TNS/R native object files that can then be used as a statically linked library by the `nld` linker on any platform where `nld` runs
- Is accessed by selecting the Tandem ar command on the Tools menu
- Can be called as a stand-alone tool from a make file or from a command prompt

The `ar` tool is also available with ETK, supporting either TNS/R or TNS/E files produced by the `ld` or `eld` linkers.

Running and Debugging C and C++ Programs

Running Programs in the Guardian Environment

When you run a C or C++ program, the NonStop OS creates a new process from the program file you specify. This new process passes through three phases of execution:

1. Program initialization: the C and C++ libraries and CRE perform startup tasks.
2. Program execution: the program controls the flow of execution.
3. Program termination: the C and C++ libraries and CRE perform shutdown tasks.

To run a C or C++ program in the Guardian environment, you use the command interpreter RUN command. This diagram shows the general form of the RUN command. For more detail, see the *TACL Reference Manual*.

```
[RUN] program-file [ / run-options / ] [ args-list ]
```

program-file

is the name of the C or C++ object file you want to run.

run-options

is a comma-separated list of options to the RUN command. Note that this list is enclosed by slashes (/). Two of the most frequently used RUN options are IN and OUT.

IN *file-name*

specifies the standard input file (`stdin`) for the new process. If you do not include the IN option, the new process uses the command interpreter's default input file, which is usually your home terminal.

OUT file-name

specifies the standard output file (`stdout`) for the new process. If you do not include the OUT option, the new process uses the command interpreter’s default output file, which is usually your home terminal.

args-list

is a space-separated list of additional arguments to the C or C++ program you are running. Note that you separate these arguments with spaces, not commas.

Usage Guidelines

- To run in the Guardian environment a native object file that was compiled in the OSS environment, you must set the file code to 700 for a TNS/R object file, 800 for a TNS/E object file, or 500 for a TNS/X object file after copying the file to the Guardian file system.

  For example, to set the file code of an object file to 700, enter this at a TACL prompt:

  ```
  > FUP ALTER filename, CODE 700
  ```

To determine the file code of an object file, enter:

```
> FUP INFO filename
```

- You can group several words into a single args-list argument by enclosing them in quotation marks; for example:

```
5> RUN invite /OUT $s.#hold/ "Bruce and Rob"
```

- To include a quotation mark as part of a quoted argument, use two quotation marks; for example:

```
6> RUN findstr /IN myfile/ "Refer to ""Running a *"""
```

Running Programs in the OSS Environment

To run a C or C++ program from the OSS environment, enter the program file name at the OSS shell prompt. You can also use the `run` command to run a process with HPE specific attributes. For more details, see the `run(1)` reference page, available either online or in the *Open System Services Shell and Utilities Reference Manual*.

Program Initialization

Your program begins execution when the operating system transfers control to your program's object code. Before executing the code, however, the C run-time library initializes its run-time environment. The C run-time library also calls a CRE initialization function.

The CRE initialization function establishes the CRE's internal data structures, I/O model, and so forth, in addition to shared facilities such as the user data heap. After the CRE has established its environment and set up shared facilities, it calls a language-specific initialization function for each language that is represented by a routine in your program, except TAL and pTAL. Each language-specific initialization function sets up its data structures and file I/O model for the language that it supports.

When the CRE completes initialization, it returns control to the C run-time library. The run-time library completes its own initialization and returns control to your main function, which begins executing the instructions in the program's object code.

For Guardian processes, the C run-time library performs several additional initialization tasks during the process startup phase, including:

- Processing the startup message sent by the command interpreter
- Processing any command interpreter PARAM or ASSIGN messages
- Opening the standard input, output, and error files: stdin, stdout, and stderr
- Invocation of the constructors for global and static variables in C++

As it performs these tasks, the C run-time library acquires detailed information regarding the environment in which the process is executing. The library saves this information in the argument array. The argument array contains program and argument information extracted from the RUN command—namely, *program-file* and each of the arguments in *args-list*.

For Guardian processes, the library also saves information in the environment array. The environment array contains environment parameters from PARAM messages.

The Guardian C run-time library includes six functions that allow the retrieval of the process startup message, the PARAM message, and the ASSIGN messages. For more details, see **Retrieving Startup Information** on page 430.

The Standard Input, Output, and Error Files

In the Guardian environment, the CRE automatically opens three standard files: stdin, stdout, and stderr. You can suppress the automatic opening of these files with the `NOSTDFILES` pragma.

In the OSS environment, the standard files are controlled by the OSS file system, not the CRE. For information on using the standard files in the OSS environment, see the *Open System Services Programmer's Guide*.

The three standard files for ANSI-model I/O are associated to physical files:

- stdin denotes the physical file specified by the IN option of the RUN command. If you do not use the IN option, stdin denotes the command interpreter's default input file, which is usually your home terminal.
- stdout denotes the physical file specified by the OUT option of the RUN command. If you do not use the OUT option, stdout denotes the command interpreter's default output file, which is usually your home terminal.
- stderr denotes the physical file specified in an ASSIGN STDERR command. If you do not use the ASSIGN command, stderr denotes the command interpreter's default output file, which is usually your home terminal.

The IN and OUT options of the RUN command were described in **Running Programs in the Guardian Environment** on page 427.

The ASSIGN command you use to specify the standard error file has the form:

```
ASSIGN STDERR, file-name
```

where *file-name* is a valid file name representing a physical file that the C compiler can access as a text-type logical file. Note that you must enter this ASSIGN command before you run your C program.

Invocation of Constructors for Global and Static Variables

During the startup of a C++ program, the constructors for global and static variables are invoked. Global and static variables have storage class `static`, which means that they retain their values throughout the execution of the entire program. All global variables have storage class `static`. Local variables and class members can be given storage class `static` by explicit use of the `static` storage class specifier.

Accessing Environment Information

You can access the environment information saved by the C library during program startup:

- By calling the `getenv()` library function
- By declaring parameters to the function `main()`

The `getenv()` function enables your program to access the environment array. The parameters to `main()` enable your program to access both the environment and argument arrays. See the *Guardian TNS C Library Calls Reference Manual*, *Guardian Native C Library Calls Reference Manual*, or the *Open System Services Library Calls Reference Manual*. For more details on how to use the `getenv()` function; the next subsection shows you how to declare parameters to `main()`.

Parameters to the Function main

C enables you to declare up to three parameters to your program's main function.

```
int main(int argc, char *argv[], char *env[]);
```

argc

is an integer value specifying the number of elements in the argument array *argv*.

argv

is the argument array. Each element (except for the last) points to a null-terminated string. *argv*[0] points to the fully qualified name of the executing program. Each of the elements *argv*[1] through *argv*[*argc*-1] points to one command-line argument; *argv*[*argc*] has the pointer value NULL.

env

is the environment array. Each element (except for the last) points to a null-terminated string containing the name and value of one environment parameter. The last element has the pointer value NULL.

When declaring parameters to the function `main()`, note:

- The parameters to `main()` are optional; you can declare no parameters, just *argc* and *argv*, or all three parameters. You cannot declare *env* alone.
- The identifiers *argc*, *argv*, and *env* are simply the traditional names of three parameters to `main()`; you can use identifiers of your own choosing.
- The elements of the environment array *env* point to strings of the form:

  ```
  "param-name=param-value"
  ```

  where *param-name* is the name of the environment parameter and *param-value* is its value.
- The `putenv()` function sets the value of a parameter in the environment array, and the `getenv()` function retrieves the value of a parameter in the environment array.

Use *env* and `putenv()` with caution, however. The environment parameter of `main()` is not updated when the environment array is enlarged or moved. Therefore, using the `env` parameter after invoking `putenv()` might result in inaccurate results. If you need to modify and examine the environment array, you should use the extermal variable `environ`, which always has the correct value.

Retrieving Startup Information

The Guardian C run-time library includes six functions that allow the retrieval of the process startup message, the PARAM message, and the ASSIGN messages. You can call these functions only from Guardian processes. These functions are listed in **C Functions That Retrieve Process Startup Information** table . For a detailed description of each function, see the *Guardian TNS C Library Calls Reference Manual* or the *Guardian Native C Library Calls Reference Manual*.

Table 61: C Functions That Retrieve Process Startup Information

Function	Action
`get_assign_msg()`	Retrieves a specified ASSIGN message.
`get_assign_msg_by_name()`	Retrieves the ASSIGN message that corresponds to the logical-unit name requested.
`get_max_assign_msg_ordinal()`	Determines how many ASSIGN messages are assigned to a particular process.

Table Continued

`get_param_by_name()`	Retrieves the value of the parameter that corresponds to the parameter name requested.
`get_param_msg()`	Retrieves the PARAM message.
`get_startup_msg()`	Retrieves the process startup message.

Program Termination

The program termination phase of execution begins when your process returns from the function `main()` or calls the `exit()` or `terminate_program()` library function. In either case, these occurs:

- All file buffers are flushed.
- All open files are closed.
- The destructors are invoked for global and static variables in C++.
- The process is terminated with a certain completion code, depending on what caused termination.

When your process returns from `main()` with no return value, your process completes with a completion code of 0, normal termination.

When your process calls `exit()` or `terminate_program()`, your process completes with normal or abnormal termination, depending on the completion code you assign to the *status* or *options* and *completion_code* parameters, respectively.

For active backup process pairs, the call to `exit()` stops both primary and backup processes.

Two Memory Models: Large and Small

HPE TNS C has two memory models: the large-memory model and the small-memory model. HPE TNS C++, native C, and native C++ has only the large-memory model.

Both models support large amounts of code, but the large-memory model also supports large amounts of data. You cannot mix modules compiled with different memory models; all modules in a program must be compiled for either the small-memory model or the large-memory model. **Memory Models** summarizes the characteristics of each memory model. **Memory Models** illustrates the memory models.

Table 62: Memory Models

Memory Model	TNS Process Code Space	Size of Data Space	Size of Pointer	Available in
Small	4 MB	64 KB	16-bit	Guardian environment for TNS C programs only
Large	4 MB	127.5 MB	32-bit	Guardian and OSS environments for all C and C++ programs

NOTE: The maximum code space for TNS/R and TNS/E code (irrespective of language) is 256MB, from 0x70000000 to 0x80000000. Refer to the *HPE NonStop S-Series Server Description Manual* for a description of how this space is allocated.

HPE strongly recommends that your TNS C programs use the large-memory model.

Small-Memory Model

The small-memory model uses 16-bit addressing and stores all data in the user data space, which can contain up to 64 KB.

The primary global area contains global scalar variables, scalar variables that are local to a function but are declared as having storage class `static`, and pointers to global aggregates that reside in the secondary global area. (A scalar type is a character, an integer, an enumeration, or a floating point type. An aggregate type is an array or struct.)

The heap area is reserved for dynamic memory. The heap is managed by the `calloc()`, `free()`, `malloc()`, and `realloc()` library functions.

The stack area is reserved for the run-time stack. The stack area stores local variables. Its size is 64 KB minus the size of the heap and the global area.

Large-Memory Model

The large-memory model uses 32-bit addressing. It stores the heap and the global and static aggregates in a single extended memory segment. Here, static aggregate refers to an aggregate that is declared with the `static` storage class specifier. Both global and static aggregates have storage class `static`, which means that they retain their values throughout the execution of the entire program.

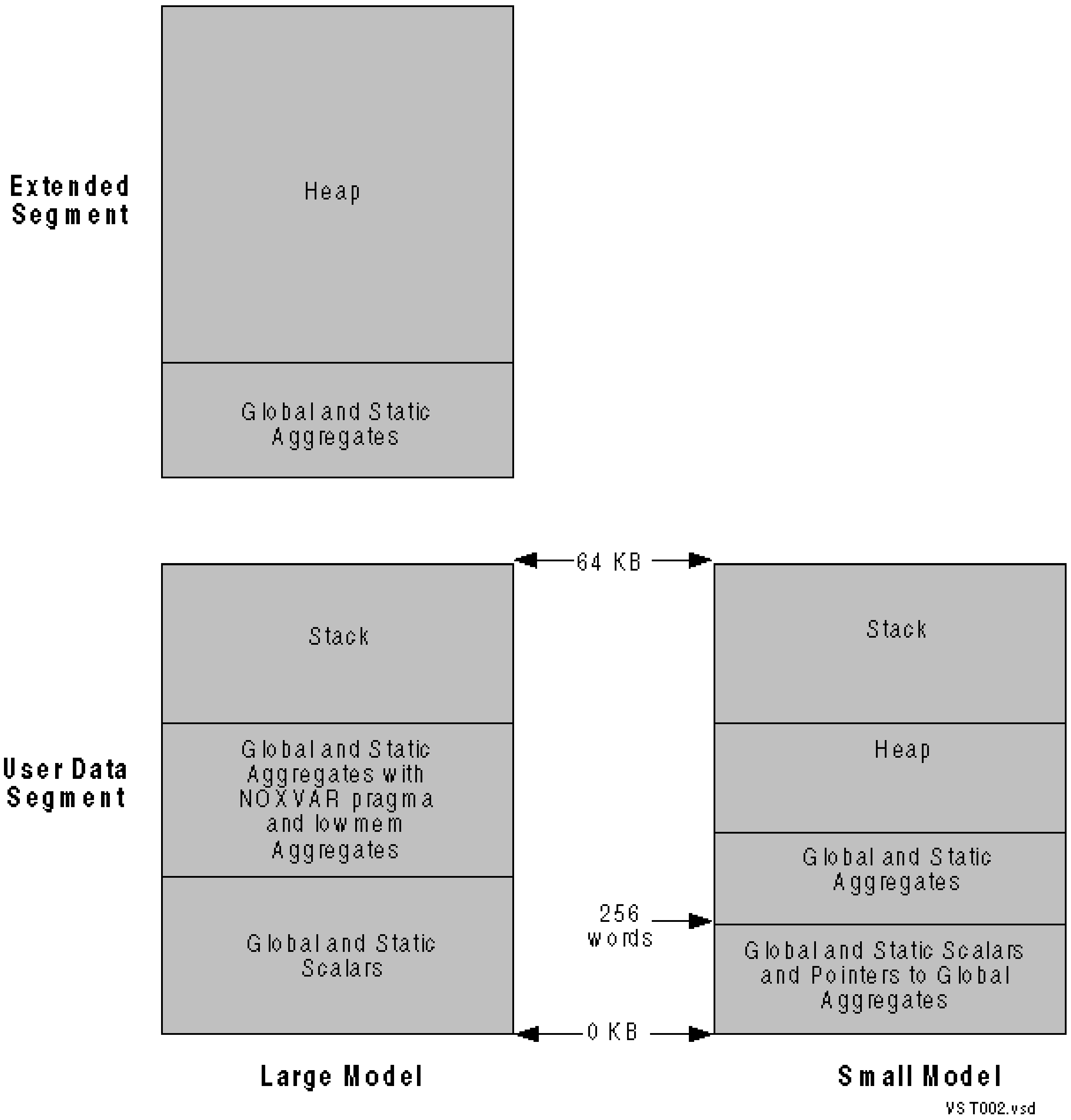
Extended Segment
Heap
Global and Static Aggregates
User Data Segment
64 KB
Stack
Global and Static Aggregates with NOXVAR pragma and lowmem Aggregates
Global and Static Scalars
Stack
Heap
Global and Static Aggregates
256 words
Global and Static Scalars and Pointers to Global Aggregates
0 KB
Large Model
Small Model
VST002.vsd

Figure 1: Memory Models

The size of the extended segment is the sum of the heap size you specify during compilation or binding and the sizes of all global and static aggregates. The physical limit of the extended segment is 127.5 MB; the practical limit depends on paging rates and disk space. If one of the memory-allocation functions (`calloc()`, `malloc()`, or `realloc()`) cannot allocate a block of the requested size from the heap, it automatically attempts to enlarge the extended segment.

The stack, global, and static aggregates that have been specified with the `NOXVAR` pragma and global and static scalars are stored in the user data segment. The user data segment is limited to a total of 64 KB.

Buffers that are used in certain Guardian system procedure calls must be stored in the user data segment so that they will be 16-bit addressable. See the description of pragma **XVAR** on page 334, for a discussion of the `XVAR` and `NOXVAR` pragmas and how to use them to specify whether global and static aggregates are stored in the extended segment or the user data segment.

Two Data Models: 16-Bit and ILP32

This section applies to TNS C/C++ only.

The data model determines the size of the type `int`. You cannot mix modules compiled with different data models; all modules in a program must be compiled for either the 16-bit data model or the ILP32 data model.

Data Models summarizes the characteristics of each data model.

Table 63: Data Models

Data Model	Size of Type int	Available in
16-bit	16 bits	Guardian environment for TNS C and C++ programs only
ILP32 or wide	32 bits	Guardian and OSS environments for all C and C++ programs

For portability and compatibility with other C environments, HPE strongly recommends that you write programs using the 32‑bit data model. For conversion details, see **Converting Programs to the ILP32 Data Model** on page 435.

Selecting Memory and Data Models

This section applies to TNS C/C++ only.

In the Guardian environment, you can select from the small-memory or large-memory model and the 16-bit or ILP32 data model. In the OSS environment, only the large-memory model and ILP32 data model are available.

For portability and compatibility with other C environments, HPE strongly recommends that you write your programs using the large-memory model and the 32‑bit data model. **Relationship Between Memory Models and Data Models** summarizes the relationship between memory models and data models.

Table 64: Relationship Between Memory Models and Data Models

Memory Model	Data Model	C Compiler Pragmas	Size of Pointer	Size of Type int

Small	16-bit	`NOXMEM`
		,
		`NOWIDE`
			16-bit	16 bits

Small	ILP32 or wide	Not available	16-bit	32 bits

Large	16-bit	`XMEM`
		,
		`NOWIDE`
			32-bit	16 bits

Large	ILP32 or wide	`XMEM`
		,
		`WIDE`
			32-bit	32 bits

NOTE: The small-memory model and the ILP32 data model cannot be selected together.

The default memory and data models used by the C compiler are determined by the environment in which the C compiler runs and the `SYSTYPE` pragma setting, as shown in **Default Memory and Data Models** . (The `SYSTYPE` pragma determines whether a program's target is the NonStop environment.) To change the default settings, specify any valid combination of the `XMEM`, `NOXMEM`, `WIDE`, and `NOWIDE` pragmas.

Table 65: Default Memory and Data Models

Compiler Environment	SYSTYPE Pragma	Memory Model	Data Model	Pragma Settings
Guardian	GUARDIAN	Large	16-bit	`XMEM, NOWIDE`
Guardian	OSS	Large	ILP32	`XMEM, WIDE`
OSS	GUARDIAN	Large	ILP32	`XMEM, WIDE`
OSS	OSS	Large	ILP32	`XMEM, WIDE`

The C compiler running in the OSS environment supports only the large-memory model and ILP32 data model.

The memory model and data model selected determines the model-dependent library file that you bind into a program. For more details, see **Compiling, Binding, and Accelerating TNS C Programs** on page 336, and **Compiling, Binding, and Accelerating TNS C++ Programs** on page 351.

Converting Programs to the ILP32 Data Model

HPE strongly recommends that you convert your TNS C and C++ programs from the 16‑bit data model to the ILP32 (or wide) data model. The native C and C++ compilers do not support the 16-bit data model. This list provides guidelines for this conversion using the TNS C compiler and Cfront:

- Compile your program using the `STRICT` pragma.
- Ensure that the type of a function call argument matches the defined type of its associated parameter. The compiler issues this warning message for argument-parameter mismatches:

  ```
  Warning 86: argument "name" conflicts with formal definition
  ```

- Write function prototypes for all user-written functions that don't have prototypes. The compiler issues this warning message for function calls that don't have corresponding function prototypes:

  ```
  Warning 95:  prototype function declaration not in scope: "function-name"
  ```

- Ensure that the formal and actual parameters of pointer types are matched. The compiler issues this warning message if pointers do not match:

  ```
  Warning 30:  pointers do not point to same type of object
  ```

  For example:

  ```
  int func1(short *);
  ```

  In the 16‑bit data model and the large-memory model, you can pass to `func1` a pointer of type `short` or `int` and get the correct results. In the ILP32 data model, you can pass to `func1` only a pointer of type `short`; a pointer of type `int` generates incorrect results.

Parameter mismatch is most often an issue for Guardian system procedures and external TAL routines.

- Ensure that literals do not cause type mismatches.

  ```
  #include <cextdecs(MONITORCPUS)>
  ...
  short get_cpu_number;
  MONITORCPUS(0x8000 >> get_cpu_number);
  ```

  In the ILP32 data model, if `get_cpu_number` is equal to zero, an arithmetic overflow occurs because the compiler generates code to convert an unsigned 32-bit integer to a 16-bit signed integer. Note that `cextdecs` does not use the type `unsigned short` .

- Avoid using the type `int` in your program, if possible. Use type `long` or `short` instead. However, if you want to keep your program data-model independent, you cannot avoid using type `int` completely. For example, C library calls, bit-fields, TCP/IP sockets library functions, and Guardian system procedures might require type `int`.

Debugging C and C++ Programs

To debug C and C++ programs, you can use several debuggers:

- Debug on TNS/R systems
- Inspect on TNS/R, TNS/E, or TNS/X systems
- Native Inspect on TNS/E and TNS/X systems and from within NSDEE on the PC
- The Visual Inspect debugger (TNS/E only)

These subsections introduce some of the features of each debugger.

Debug

Debug provides machine-level process debugging; that is, it provides access to a process in terms of code and data addresses, hardware registers, and other machine-level quantities. To use Debug, you should have a thorough understanding of the HPE NonStop architecture as described in the system description manual that corresponds to your NonStop system, such as the *NonStop S-Series Server Description Manual*.

For more details about Debug, see the *Debug Manual*.

Inspect

The Inspect symbolic debugger provides both machine-level and source-level process debugging.

If you compile your TNS or TNS/R native program using the `SYMBOLS` pragma, you can use the source-level mode of the Inspect debugger to access your process in terms of variables, functions, statements, and other source-level entities. In addition, if you have a thorough understanding of the HPE NonStop architecture, you can use the machine-level mode of the Inspect debugger to access hardware registers and other machine-level quantities.

If you compile your program using the `SAVEABEND` pragma, the system automatically creates a save file, or snapshot, of your running program if it terminates abnormally. You can use Inspect to examine the save file and diagnose the cause of the abnormal termination.

To debug PIC (Position-Independent Code) on a TNS/R system, you must use Visual Inspect. To debug any native code file on a TNS/E system, you must use Visual Inspect or Native Inspect. To debug any native code file on a TNS/X system, you must use Native Inspect or NSDEE.

You can use Inspect on both TNS and TNS/R programs. You cannot use Inspect on TNS/E or TNS/X programs.

For more details about the Inspect debugger, see the *Inspect Manual*.

Native Inspect

Native Inspect is a symbolic debugger for TNS/E and TNS/X systems and is based upon the open-source GNU `dbg` utility. Native Inspect provides both machine-level and source-level process debugging.

If you compile your TNS/E or TNS/X native programs using the `SYMBOLS` pragma, you can use the source-level commands to access your process in terms of variables, functions, statements, and other source-level entities. In addition, if you have a thorough understanding of the HPE NonStop architecture, you can use the machine-level commands to access registers, code and data segments, stack, and other machine-level entities.

If you compile your program using the `SAVEABEND` pragma, the system automatically creates a save file, or snapshot, of your running program if it terminates abnormally. You can use Native Inspect to examine the save file and diagnose the cause of the abnormal termination.

For more details about Native Inspect, see the *Native Inspect Manual.*

Native Inspect is also integrated with NSDEE when you have installed NSDEE Core with Debugging (see **NSDEE** on page 419).

Visual Inspect Symbolic Debugger

Visual Inspect is an optional HPE PC-based (GUI) symbolic debugger designed to work in complex distributed environments. Visual Inspect provides only source-level debugging for TNS/E servers executing in the NonStop environments.

Visual Inspect is a client-server application; the server runs on the HPE NonStop platform, and the client runs on the PC. Visual Inspect is also integrated into the HPE Enterprise Tool Kit on the PC.

As when using the Inspect or Native Inspect symbolic debuggers, you must use the `SYMBOLS` pragma to enable the inclusion of symbol information in your object file and you should not strip the executable file using a linker or the `strip` utility. You can also use the `SAVEABEND` pragma to create a save file if your program terminates abnormally.

For more details about enabling and using Visual Inspect, see the online help for Visual Inspect.

Debugging an Instrumented Application

If you use the **CODECOV** on page 214 command line option to direct a TNS/E or TNS/X C or C++ compiler to generate instrumented object code, the Code Coverage Utility connects to the application program as a debugger to read its memory, and so on. This causes all the debug requests to wait until the Code Coverage Utility (the active debugger) detaches from the application. The Code Coverage Utility only detaches after the instrumented application stops.

Therefore, if you must debug an instrumented application, it should be started in debug mode by using the TACL RUND or RUNV commands.

TNS C Compiler Messages

This chapter presents the error and warning messages that can occur during the compilation of a C program using the TNS C compiler.

Types of Compiler Messages

The TNS C compiler scans the source code line by line and notifies you of an error or potential error by issuing an error or warning message:

- An error message indicates an error that you must correct before C can successfully compile the source code.
- A warning message indicates a potential error condition and an area of the source code that you should check.

When it issues a message, the compiler displays:

- Either ERROR or WARNING to indicate the type of message
- The message number
- The text of the message

In addition, the compiler displays the line number of the source line that caused the message and, in the case of include files, the file name.

1

```
invalid preprocessor command
```

Cause

The preprocessor encountered an unrecognized element that was not a valid preprocessor token. For example, the C compiler generates this error if it encounters a # (number sign) character at the start of an input line and this text is not a preprocessor directive.

2

```
unexpected end of file
```

Cause

The compiler encountered the end of an input file when it expected more data. This error can occur on an included file or the original source file. In many cases, correction of a previous error will eliminate this error.

3

```
file not found, file in use, security violation, or Guardian error = "error-number"
```

Cause

The compiler did not find the file with the name specified on an `#include` directive.

4

```
invalid lexical token
```

Cause

The compiler encountered an unrecognized element that was not a valid lexical construct (such as an identifier or one of the valid expression operators). This error can occur if the compiler detected control characters or other illegal characters in the source file. This error can also occur if you specified a preprocessor directive with the number sign not in the first position of an input line.

5

`number of actual and formal parameters in a macro are not matched, macro is not expanded`

Cause

You specified a preprocessor `#define` macro with the wrong number of arguments.

6

`line buffer overflow`

Cause

Expansion of a `#define` macro caused the compiler's line buffer to overflow. This error can occur if more than one lengthy macro appeared on a single input line.

7

`file stack full`

Cause

You exceeded the maximum depth of `#include` nesting; the compiler supports `#include` nesting to a maximum depth of 16.

8

`invalid conversion`

Cause

You specified an invalid arithmetic or pointer conversion. This error usually results from attempts to convert a scalar to an aggregate or function.

9

`undefined identifier` *name*

Cause

The named identifier was undefined in the scope where it appeared; that is, you did not previously declare it. This message is generated only once; subsequent encounters with the identifier assume that it is of type `int` , which can cause other errors.

10

`invalid subscript expression`

Cause

The compiler detected an error in the expression—presumably a subscript expression—following the left bracket character. This error can occur if the expression in brackets is null (not present).

11

`string too large or not terminated`

Cause

The length of a string constant exceeded the maximum allowed by the compiler (256 bytes). This error occurs if you omit the closing double quote when specifying the string.

12

`invalid structure reference`

Cause

The compiler did not accept the expression preceding the period or arrow operator as a structure or pointer to a structure. This error can occur even for constructions that are accepted by other compilers.

13

`member name missing`

Cause

The compiler did not find an identifier following the period or arrow operator.

14

`undefined member` *`member-name`*

Cause

The indicated identifier was not a member of the structure or union to which the period or arrow operator referred. This error can occur for constructions that are accepted by other compilers.

15

`invalid function call`

Cause

You did not implicitly or explicitly declare the identifier preceding the function call operator as a function.

16

`invalid function argument`

Cause

You specified an invalid function argument expression following the `(` function call operator. This error can occur if you omit an argument expression.

17

`too many operands`

Cause

During expression evaluation, the compiler encountered the end of an expression, but more than one operand was still awaiting evaluation. This error can occur if you omit an expression.

18

`unresolved operator`

Cause

During expression evaluation, the compiler encountered the end of an expression, but more than one operand was still pending evaluation. This error can occur if you omit an operand for a binary operation.

19

`unbalanced parentheses`

Cause

The number of opening and closing parentheses in an expression was not equal. This error can also occur if a macro was poorly specified or improperly used.

20

```
invalid constant expression
```

Cause

The compiler encountered an expression that did not evaluate to a constant in a context that required a constant result. This error can occur if one of the operators not valid for constant expressions was present.

21

```
illegal use of aggregate
```

Cause

An identifier declared as a structure or union was encountered in an expression without being properly qualified by an aggregate reference.

22

```
structure used as function argument
```

Cause

An identifier declared as a structure or union appeared as a function argument without the preceding address-of operator. Although you can pass a structure by value, this warning reminds you that it is inefficient to do so.

23

```
invalid use of conditional operator
```

Cause

You used the conditional operator erroneously. This error can occur if the question mark was present but the colon was not found when expected.

24

```
pointer operand required
```

Cause

The context of the expression required a pointer operand. This error can occur if the expression following the `*` indirection operator did not evaluate to a pointer.

25

```
lvalue required
```

Cause

The context of the expression required an operand to be an lvalue. This error can occur if the expression following the address-of operator was not an lvalue, or if the left side of an assignment expression was not an lvalue.

26

```
arithmetic operand required
```

Cause

The context of the expression required an operand to be arithmetic (not a pointer, function, or aggregate).

27

`arithmetic or pointer operand required`

Cause

The context of the expression required an operand to be either arithmetic or a pointer. This error can occur for the logical `OR` and logical `AND` operators.

28

`missing operand`

Cause

During expression evaluation, the compiler encountered the end of an expression but not enough operands were available for evaluation. This error can occur if you improperly specified a binary operation.

29

`invalid pointer operation`

Cause

You specified an operation—such as an arithmetic operation other than addition or subtraction—that was invalid for pointer operands.

30

`pointers do not point to same type of object`

Cause

In an assignment statement defining a value for a pointer object, the expression on the right side of the assignment operator did not evaluate to a pointer of the exact same type as the pointer object being assigned; that is, it did not point to the same type of object. This warning also occurs if you assign a pointer of any type to an arithmetic object. Note that the same message becomes a fatal error if generated for an initializer expression.

31

`integral operand required`

Cause

The context of an expression required an operand to be integral; that is, one of the integer types (`char` , `int` , and so on).

32

`invalid conversion specified`

Cause

The expression specifying the type name for a cast (conversion) operation or a `sizeof` expression was invalid.

33

`cannot initialize auto aggregate`

Cause

You attempted to attach an initializer expression to a structure, union, or array that was declared auto. These initializations are not allowed by the language.

34

`invalid initializer expression`

Cause

The expression used to initialize an object was invalid. This error can occur for a variety of reasons, including failure to separate elements in an initializer list with commas or specification of an expression that did not evaluate to a constant. This error might require some experimentation to determine its exact cause.

35

`closing brace expected`

Cause

During processing of an initializer list or a structure or union member declaration list, the compiler did not find a closing right brace. This error can occur if you specified too many elements in an initializer expression list, or if you improperly declared a structure member.

36

`unreachable code`

Cause

The compiler isolated some source code that could never be executed given the control flow of your program. An example of this situation is when a jumping statement such as `goto` or `return` unconditionally causes control to skip the source code that follows it.

37

`duplicate statement label` *`label-name`*

Cause

The compiler encountered the specified statement label more than once during processing of the current function.

38

`unbalanced braces`

Cause

In a body of compound statements, the number of opening left braces and closing right braces was not equal. This error can occur if the compiler got “out of phase” due to a previous error.

39

`invalid use of keyword`

Cause

One of the C language reserved words appeared in an invalid context (for example, as an object name).

40

`break not inside loop or switch`

Cause

The compiler detected a `break` statement that was not within the scope of a `while` , `do` , `for` , or `switch` statement. This error can occur due to an error in a preceding statement.

41

`case not inside switch`

Cause

The compiler encountered a `case` statement outside the scope of a `switch` statement. This error can occur due to an error in a preceding statement.

42

`invalid case expression`

Cause

The expression defining a case value did not evaluate to an `int` constant.

43

`duplicate case value`

Cause

The compiler encountered a `case` statement that defined a constant value already used in a previous `case` statement within the same `switch` statement.

44

`continue not inside loop`

Cause

The compiler detected a `continue` statement that was not within the scope of a `while` , `do` , or `for` loop. This error can occur due to an error in a preceding statement.

45

`default not inside switch`

Cause

The compiler encountered a `default` statement outside the scope of a `switch` statement. This error can occur due to an error in a preceding statement.

46

`more than one default`

Cause

The compiler encountered a `default` statement within the scope of a `switch` statement in which a preceding `default` statement had already been encountered.

47

`while missing from do statement`

Cause

Following the body of a `do` statement, the compiler did not find the `while` clause. This error can occur due to an error within the body of the `do` statement.

48

`invalid while expression`

Cause

The expression defining the looping condition in a `while` or `do` loop was not present. If you are trying to define an infinite loop, you must supply the constant 1 as the looping condition.

49

`else not associated with if`

Cause

The compiler detected an `else` keyword that was not within the scope of a preceding `if` statement. This error can occur due to an error in a preceding statement.

50

`label missing from goto`

Cause

The compiler did not find a statement label following the `goto` keyword.

51

`label name conflict` *`identifier`*

Cause

The indicated identifier, which appeared in a `goto` statement as a statement label, was already defined as an object within the scope of the current function.

52

`invalid if expression`

Cause

The expression following the `if` keyword was null (not present).

53

`invalid return expression`

Cause

The compiler could not convert the expression following the `return` keyword to the type of the value returned by the function. This error can occur if the expression specified a structure, union, or function.

54

`invalid switch expression`

Cause

The expression defining the value for a `switch` statement did not define an `int` value or a value that could be legally converted to `int` .

55

`no case values for switch statement`

Cause

The statement defining the body of a `switch` statement did not contain at least one `case` statement.

56

`colon expected`

Cause

The compiler did not find an expected colon. This error can occur if you improperly specified a `case` statement, or if you omitted the colon following a label or prefix to a statement.

57

`semicolon expected`

Cause

The compiler did not find an expected semicolon. This error generally means that the compiler completed the processing of an expression but did not find the statement terminator. This error can occur if you included too many closing parentheses, if you omitted the statement terminator, or if the expression were otherwise incorrectly formed.

58

`missing parenthesis`

Cause

The compiler did not find a parentheses required by the syntax of the current statement (as in a `while` or `for` loop). This error can occur if you incorrectly specified the enclosed expression, causing the compiler to end the expression early.

59

`invalid storage class`

Cause

This error can occur for several reasons:

- In processing external data or function definitions, the compiler encountered a storage class—such as `auto` or `register` —invalid for that declaration context.
- If, due to preceding errors, the compiler began processing portions of the body of a function as if they were external definitions.
- A storage class other than `register` appeared in the declaration of a parameter.

60

`incompatible aggregate types`

Cause

An attempt was made to assign an aggregate of one type to an aggregate of another type.

61

`undefined structure tag` *`tag-name`*

Cause

You did not previously define the indicated structure or union tag; that is, the members of the aggregate were unknown.

62

`structure/union type mismatch`

Cause

The compiler detected a structure or union tag in the opposite usage from which it was originally declared; that is, a tag originally applied to a structure has appeared on an aggregate with the union specifier.

63

`duplicate declaration of item` *`item`*

Cause

The compiler encountered a `default` statement within the scope of a `switch` statement in which a preceding `default` statement had already been encountered.

64

`structure contains no members`

Cause

A declaration of the members of a structure or union did not contain at least one member name.

65

`invalid function definition`

Cause

An attempt was made to define a function body when the compiler was not processing external definitions. This error can occur if a preceding error caused the compiler to “get out of phase” with respect to declarations in the source file.

66

`invalid array limit expression`

Cause

The expression defining the size of a subscript in an array declaration did not evaluate to a positive `int` constant. This error can also occur if a zero length was specified for the leftmost subscript.

67

`illegal object`

Cause

A declaration specified an illegal object, which can include functions that return aggregates (arrays, structures, or unions) and arrays of functions.

68

`illegal object for structure`

Cause

A structure or union declaration included an object declared as a function. (An aggregate, however, can contain a pointer to a function.)

69

`structure includes instance of self`

Cause

The structure or union whose declaration was just processed contains an instance of itself. This error can occur if the `*` indirection operator is forgotten on a structure pointer declaration, or if the structure actually contains an instance of itself.

70

`illegal object for formal`

Cause

You declared a parameter illegally in a function declaration. For example, if you declare a formal parameter to be a function instead of a pointer to a function, you get this error.

71

`formal declaration error` *`identifier`*

Cause

You declared an object before the opening brace of a function, but you did not include it in the list of parameter names enclosed in parentheses following the function name.

72

`external item attribute mismatch`

Cause

You declared an external item with attributes that conflict with a previous declaration. This error can occur if a function was used earlier, as an implicit `int` function, and was then declared as returning some other type of value. You must declare functions that return a value other than `int` before they are used so that the compiler is aware of the type of the function value.

73

`declaration expected`

Cause

While processing object declarations, the compiler did not find another line of declarations that it expected. This error can occur if a preceding error caused the compiler to “get out of phase” with respect to declarations.

74

`initializer data truncated`

Cause

More initializer data was provided than could be used for the declared size of the array.

75

`invalid sizeof expression`

Cause

You supplied an invalid operand in a `sizeof` expression.

76

`left brace expected`

Cause

The compiler did not find an opening left brace in the current context. This error can occur if you omitted the opening brace on a list of initializer expressions for an aggregate.

77

`identifier expected`

Cause

In processing a declaration, the compiler expected to find an identifier that was to be declared. This error can occur if you improperly specify the prefixes or postfixes (asterisks, parentheses, and so on) in a declaration, or if you list a sequence of declarations incorrectly.

78

`undefined statement label` *`label-name`*

Cause

The indicated statement label was referred to in the most recent function in a `goto` statement, but no definition of the label was found in that function.

79

`duplicate enumeration value`

Cause

You have attempted to give the same integer value to two identifiers that denote enumeration constants within the same enumeration type.

80

`invalid bit field`

Cause

The number of bits specified for a bit field was invalid. Note that the compiler does not accept bit fields that are exactly the length of a machine word; you must declare these as ordinary `int` or `unsigned` objects.

81

`preprocessor symbol loop`

Cause

The current input line contained a reference to a preprocessor symbol that was defined with a circular definition or loop.

82

`maximum object/storage size exceeded`

Cause

The size of an object exceeded the maximum legal size for objects in its storage class. Alternatively, the last object declared caused the total size of declared objects for that storage class to exceed the maximum.

83

`reference beyond object size`

Cause

An indirect pointer reference (usually a subscripted expression) used an address beyond the size of the object used as a base for the address calculation. This error generally occurs when you refer to an element beyond the end of an array.

84

`redefinition of preprocessor symbol` *`symbol-name`*

Cause

The compiler encountered a `#define` directive for an already defined symbol. The second definition takes precedence, but it requires an additional `#undef` directive before the symbol is truly undefined.

86

`argument` *`name`* `conflicts with formal definition`

Cause

The type of the specified argument conflicts with the defined type of its associated parameter.

87

`argument count incorrect`

Cause

The number of arguments in the function call disagrees with the number of arguments in the function prototype.

88

`argument type incorrect`

Cause

The type of an argument in the function call disagrees with the type specified in the function prototype.

89

`constant converted to required type`

Cause

A constant in the function call did not have the type specified in the function prototype. The constant has been converted to the appropriate type.

90

`invalid argument type specifier`

Cause

The parameter type specifier within the function prototype is an invalid type specifier.

91

`illegal void operand`

Cause

You specified a `void` operand where it was invalid.

92

`statement has no effect`

Cause

The statement does nothing. In this common instance, the statement `f;` is meant to be a call to `f` , a function without parameters. Programmers accustomed to TAL often omit the parentheses from such a function call.

93

`no reference to identifier` *`name`*

Cause

The identifier was not referenced at any point while it was in scope.

94

`uninitialized auto variable` *`object-name`*

Cause

The auto object was used on the right side of an expression before it was initialized, assigned to, or had its address taken.

95

`prototype function declaration not in scope:` *`func-name`*

Cause

The compiler encountered a reference to a function that does not have a function-prototype declaration in scope.

96

`function declaration not in prototype form:` *`func-name`*

Cause

The compiler encountered a function declaration or definition that does not use function-prototype syntax.

Cause

This warning occurs only when you have specified the `STRICT` pragma.

97

`formal parameter was not declared:` *`name`*

Cause

The compiler encountered a nonprototype function definition that does not define the type of one of its parameters. Consequently, that parameter defaults to type `int` . This warning occurs only when you have specified the `STRICT` pragma.

98

`no type was specified for` *`object-name`*

Cause

The compiler encountered an object declaration that did not specify the object’s type. Consequently, the object defaults to type `int` .

This warning occurs only when you have specified the `STRICT` pragma.

99

`no cast for conversion that may cause loss of significance`

Cause

The compiler encountered a conversion from a “higher” type to a “lower” type that did not explicitly specify the conversion using the cast operator.

This warning occurs only when you have specified the `STRICT` pragma.

100

```
include file loop: file-name
```

Cause

An `#include` directive specifies a file that is currently being included in the compilation. For example, if SOURCE1 includes SOURCE2, an attempt by SOURCE2 to include SOURCE1 would generate this message.

101

```
illegal object with void type: object-name
```

Cause

The declaration of an object specified `void` as the object's type. This usually occurs when you forget to precede the object name with an asterisk (*) to specify the type pointer to `void` .

102

```
no room for static data: object-name
```

Cause

A program being compiled using the small-memory model attempted to declare more than 32 KB of static data.

103

```
no room for automatic data: object-name
```

Cause

A function definition declared more than 32 KB of automatic data.

104

```
constant converted to smaller type and significance lost
```

Cause

A constant was converted to a type that could not represent the value of the constant.

105

```
function declaration not in scope: func-name
```

Cause

The compiler encountered a reference to a function that does not have a function declaration in scope.

This warning occurs only when you have specified the `STRICT` pragma.

106

```
variables in 16-bit addressable memory can be no more than 32,767 bytes long:
object-name
```

Cause

An object declaration specified that the size of an object in low memory was larger than 32 KB. An object resides in low memory when you compile for the small-memory model or when you compile for the large-memory model and specify the storage class `_lowmem` .

107

```
constant used as logical expression
```

Cause

You used a constant as the controlling condition of an `if` , `while` , or `do` statement.

This warning occurs only when you have specified the `STRICT` pragma.

108

`constant used as switch expression`

Cause

You used a constant as the selector expression in a `switch` statement.

This warning occurs only when you have specified the `STRICT` pragma.

110

`number is not a legal warning number`

Cause

You specified an invalid warning number in the `WARN` pragma.

111

`*pragma* directive has appeared too late in the compilation`

Cause

You used one of the pragmas that must occur at the start of the translation unit somewhere later in the translation unit.

112

`alias name for external C routine expected`

Cause

In a mixed-language program, the COBOL, FORTRAN, Pascal, or TAL procedure name is either illegal as a C identifier or conflicts with a predefined name in C. You must define an alternative procedure name using an external name. This error can also occur if the character string that defines the external procedure name is missing.

113

`argument *arg* to non-C routine is not declared lowmem`

Cause

You attempted to pass the address of an object in extended memory. When using the large-memory model, you must allocate user data space for arrays and structures that are arguments to a non-C procedure using the `_lowmem` storage class specifier.

114

`copy of structure greater than 65,535 bytes not supported`

Cause

You attempted to assign or pass as an argument a structure larger than 64 KB. The C compiler does not support copying of structures of this size.

115

`section name was not found in the include file: *name*`

Cause

There is a mismatch between the file name and section you specified in the `#include` directive and the names in the `SECTION` pragmas of the file.

117

`C compiler is out of heap space`

Cause

This error occurs only with an extremely large C compilation unit. Break the large unit into two or more smaller units, and then use Binder to link them after compilation.

118

`illegal function definition`

Cause

The parameter identifiers are missing from a function definition.

119

`function was declared previously with old-style declaration`

Cause

After encountering a nonprototype declaration of a function, the compiler subsequently encountered a prototype declaration or definition of the same function.

120

`argument mismatch with previous declaration of function`

Cause

A subsequent declaration or definition of a previously declared function specifies a different type for one of the parameters.

121

`pragma directive specified but there is no main() function`

Cause

You specified the `RUNNABLE` pragma in a program that contains no main function.

122

`invalid object filename specified`

Cause

You did not specify a valid file name for the object file in the compilation command.

123

`specified heap size is too big`

Cause

The value provided in the `HEAP` pragma is too large. This usually occurs when compiling programs for the small-memory model.

124

`directive is valid only on the command line:` *`pragma`*

Cause

You used one of the pragmas that must occur on the command line in the translation unit.

125

`directive is valid only as a pragma in the program: *pragma*`

Cause

You used one of the pragmas that must occur in the translation unit on the command line.

126

`comment is terminated by EOF - comment started on line: *line*`

Cause

The compiler encountered the end of an included file or the translation unit before encountering the end of an opened comment.

127

`input file is not a supported type`

Cause

You did not specify a type of input file in the compilation command.

128

`extptr may only be used for non-C formal parameter pointer declarations`

Cause

You used `extptr` outside of a non-C interface declaration, or you used it when declaring a nonpointer parameter in a non-C interface declaration.

129

`obsolete TAL declaration syntax - use prototype syntax`

Cause

You used non-prototype syntax in a TAL interface declaration.

130

name `is not a SQL variable`

Cause

The C object you used in an embedded SQL string was not declared within an SQL Declare Section.

131

`variable missing from SQL statement`

Cause

The compiler encountered a colon (:) in an embedded SQL string, but the colon was not followed by a valid C identifier.

132

name `is not of a supported variable type in SQL statements`

Cause

The object you specified as a variable in an embedded SQL string is not of a supported type.

133

`branch to label` *`label-name`* `in inner block that contains aggregates not supported`

Cause

The C compiler does not support `goto` statements that transfer control into a block that contains declarations of aggregates.

134

`name` `was not declared by including the appropriate header file`

Cause

The specified library routine was used before it was declared. The C compiler requires that you include the header for a library routine before you use the routine.

135

`proper header file was not used to declare library function` *`name`*

Cause

The specified library routine was declared, but not by including its header. The C language requires that you declare library routines by including the appropriate header.

136

`library function` *`name`* `has been redefined`

Cause

The compiler encountered a redefinition of the specified library routine. This error usually indicates that one of your functions has a name conflict with a library routine.

137

`name` `is an obsolete feature`

Cause

The specified feature is obsolete in this release of the C compiler.

139

`arithmetic overflow/underflow detected in constant arithmetic`

Cause

The resultant value of a constant expression exceeded the range of the object to which it was assigned.

143

`filename has been truncated to` *`file-name`*

Cause

The preprocessor encountered a file name longer than 8 characters in an `#include` directive and so truncated it to the specified name.

144

`illegal use of _cc_status, this declaration type is reserved for TAL procedure return status`

Cause

The type `_cc_status` is valid only as the function return-type in an interface declaration of a TAL procedure.

145

`illegal function pointer operation`

Cause

The compiler encountered an invalid use of a pointer to a function.

146

`missing comma between directives`

Cause

The compiler encountered a `#pragma` directive that specifies multiple pragmas, but the pragmas are not separated by commas.

147

`non-standard use of cast operator in integral constant expression`

Cause

In an integral constant expression, the compiler encountered a cast operator that converts a pointer value to an arithmetic type. The C compiler does not support this nonstandard use of the cast operator.

148

`return possibly missing`

Cause

A function with a non- `void` return type does not have a `return` statement that is unconditionally executed and that appears lexically immediately before the function's closing brace.

149

`unknown pragma directive` *`pragma`*

Cause

The compiler encountered an invalid pragma directive. This message is a warning message.

150

`#error` *`message-text-you-specify`*

Cause

The `#error` preprocessor directive allows you to force a compilation error and designate the text of the error message. This error message is printed as the message text of error 150.

151

`macro-name` `has been replaced once, no further replacement`

Cause

During the macro expansion phase, if *macro-name* is encountered and if *macro-name* is the same as that of the expanding macro, *macro-name* is not expanded, to avoid infinite looping of a recursive call.

This message is a warning message.

152

```
missing ending quote
```

Cause

The C compiler encountered a string literal that does not have an ending quotation mark (").

153

```
A ## preprocessing token shall not occur at the beginning or at the end of a
replacement list
```

Cause

A `##` preprocessor operator occurs at the beginning or at the end of a replacement list. The binary operator `##` is used in macro definitions. It allows you to concatenate two tokens, and therefore cannot occur at the beginning or at the end of the macro definition.

154

```
A # preprocessing token must be followed by a parameter
```

Cause

A `#` preprocessor operator occurs without an accompanying parameter. The unary operator # is used only with function-like macros, which take arguments. When the `#` operator precedes a formal parameter in the macro definition, the actual argument passed by the macro invocation is enclosed in quotation marks and treated as a string literal.

155

```
ambiguous directive
```

Cause

When you specify a compiler pragma, you can abbreviate the pragma name. However, you need to specify enough letters in the abbreviated pragma name to make the name unique with respect to all the other C compiler pragma names. For example, there is the `SEARCH` pragma and the `SECTION` pragma. To enter a unique abbreviation for `SEARCH` use `SEA` and for `SECTION` use `SEC` . The abbreviation `SE` for either of the pragmas is not unique. If `SE` is specified, this error message occurs.

156

```
obsolete 'tal' keyword; use '_tal' instead
```

Cause

The C compiler encountered an interface declaration for a TAL procedure that uses the keyword `tal` . Replace the keyword `tal` with the keyword `_tal` .

157

```
only unqualified filenames allowed in #include with angle brackets
```

Cause

The C compiler encountered a qualified file name in a `#include` preprocessor directive with angle brackets. For example, the directive `#include,<$a.b.c>` must be changed to `#include "$a.b.c"` .

158

```
compiler should be run in a low PIN
```

Cause

The compiler process on this system cannot run at a high PIN.

159

```
missing close parenthesis
```

Cause

The compiler encountered an expression with unbalanced parentheses.

160

```
missing comma between section names
```

Cause

The compiler encountered a list of two or more section names that are not separated by commas. The `SECTION` pragma gives a name to a section of a source file for use in a `#include` directive. If named sections are included from the same file, they must all be included in the same `#include` directive and these section names must be separated by commas as in this example:

```
#include "headerh(sectiona,sectionb,sectionc)"
```

161

```
too many commas
```

Cause

The C compiler encountered extra commas in a list of section names. When including several section names from a file, use only the minimal number of commas necessary to separate the section names. For example, this `#include` directive causes this error message.

```
#include "$system.system.cextdecs(mypid,,sqlcadisplay)"
```

162

`character` *`extra-character`* `is extra`

Cause

This is a warning message that notifies you that the compiler has found an extra character. The message specifies the character.

163

```
macro expansion buffer overflow
```

Cause

During the macro expansion phase, the C compiler’s internal buffer experienced an overflow condition. Reduce the size of the macro text.

164

```
type is not defined or a comma is missing
```

Cause

The C compiler encountered a variable whose type is not defined, or the compiler encountered a situation where a comma is missing.

In this example, if the type of the variable t is undefined, the error message is generated.

```
void f(t *x);
```

In this example, there is a comma missing between the parameter types, which generates the error message.

```
void f1(p1 p2);
```

165

```
constant for pointer function argument may only be 0 (null pointer)
```

Cause

In the absence of function prototypes, the C compiler issues this error if a pointer to a const-qualified variable is passed as an actual parameter. This error message is issued to prevent the value of a const-qualified variable from being changed.

166

```
WIDE (32 bit integers) only supported with XMEM
```

Cause

The 32-bit data model (or wide data model), which is created by compiling your program with the `WIDE` pragma, can exist only under the large memory model. You might have compiled your program using both the `WIDE` pragma and the `NOXMEM` pragma.

167

```
#endif, #else, or #elif out of order
```

Cause

The compiler encountered a sequence of `#elif` , `#else` , and `#endif` preprocessor directives that do not follow this logical sequence:

```
#elif int-constant-expression
```

possibly followed by the preprocessing directive

```
#else
```

and, finally followed by the preprocessing directive

```
#endif
```

168

```
missing #endif
```

Cause

The compiler encountered an if section that begins with `#if` and does not contain a corresponding `#endif` that marks the end of the if section.t

169

```
extra #endif
```

Cause

The compiler encountered an `if` section that contains a `#endif` preprocessing directive without a matching `#if` preprocessing directive to begin the `if` section.

170

```
character format in SQL directive was set twice; default is used
```

Cause

The default structure for a C string that is used as an SQL host variable is a null terminated array of characters. The `char_as_array` option of the `SQL` pragma directs the compiler to pass strings to SQL as an array of characters with no null terminator. The C compiler encountered two `SQL` pragmas with the `char_as_array` option specified. Therefore, the default of a C string being a null-terminated array of characters is being used.

171

```
missing end declare section directive for previous begin declare section
directive
```

Cause

In SQL, when you have an exec sql BEGIN DECLARE SECTION directive you must have a matching exec sql END DECLARE SECTION directive. The C compiler encountered an END DECLARE SECTION directive without a matching BEGIN DECLARE SECTION directive.

172

```
sqlmem directive only valid with XMEM compilation
```

Cause

When you specify the `SQLMEM` pragma, you must have previously specified the `XMEM` pragma.

173

```
illegal sqlmem option
```

Cause

When you specify the `SQLMEM` pragma, the `XMEM` and `SQL` pragmas must have been previously defined.

174

```
the argument of sizeof function has an incomplete type
```

Cause

The C compiler encountered a `sizeof` function whose argument does not have a complete type. In this example, the compiler issues this error message because the type of `x` is not completely defined.

```
struct x; ... sizeof x;
```

175

```
two storage class specifiers exist in the same declaration
```

Cause

Only one storage class specifier is allowed in each of these:

- a simple variable declaration
- an array declaration
- a pointer declaration
- a function definition or a function prototype.

The compiler encountered one of these entities that contains two storage class specifiers.

176

```
two type qualifiers exist in the same declaration
```

Cause

A simple variable, an array, a pointer variable, and the return type of a function can have only one type. The compiler encountered a declaration for one of these entities that has two types specified.

177

```
two typedef names exist in the same declaration
```

Cause

The C compiler encountered a `typedef` declaration in which two typedef names are specified. For example,

```
typedef char i j;
```

178

```
invalid signed/unsigned type
```

Cause

The C compiler encountered a type definition that is incorrectly defined with regards to the `signed` or `unsigned` attribute. For example,

```
unsigned struct x
```

179

```
incompatible type was specified
```

Cause

This error message is generated if two types are used to declare a variable. The invalid declaration can comprise two types, or a type and a typedef name, or two typedef names. An example of an invalid declaration is:

```
char int i;
```

180

```
keywords 'typedef' '_pascal' _fortran' '_cobol' '_tal' or '_unspecified'
should be the first token of a declaration statement or prototype declaration
```

Cause

The compiler encountered a `typedef` declaration or an interface declaration for a Pascal, FORTRAN, COBOL, TAL, or an unspecified language type, in which `typedef` , `_pascal` , `_fortran` , `_cobol` , `_tal` , or `unspecified` is not the first keyword in the declaration, respectively.

181

```
_lowmem can only exist with extern/static/typedef keywords in a declaration
```

Cause

The storage class specifier `_lowmem` can only exist in the same declaration with the storage class specifier `extern` or `static` . In addition, the `_lowmem` storage class specifier can be used in a `typedef` . The compiler encountered an instance in which the storage class specifier `_lowmem` was specified in a declaration along with another storage class specifier other than `extern` or `static` or with another keyword other than `typedef` .

182

```
the variable should be pointer type
```

Cause

The C compiler encountered a variable that needs to be defined as a pointer.

183

```
name has an invalid subvolume name
```

Cause

The C compiler encountered an invalid subvolume name.

184

```
format is offsetof ( type , member )
```

Cause

The call to the `offsetof` function has the wrong parameters.

185

```
first argument of offsetof is not a structure type
```

Cause

The first parameter of the `offsetof` function must be a structure or union type.

186

```
second argument of offsetof is not a member of the structure
```

Cause

The second parameter of the `offsetof` function must be a member of the structure that is the first parameter.

187

```
second argument of offsetof must not be a bit field member of the structure
```

Cause

The second parameter of the `offsetof` function cannot be a bit field member of the structure that is the first parameter of the `offseof` function.

188

```
this struct or union has no tag and no typedef name
```

Cause

The structure declaration does not have a tag. If this is a `typedef` declaration, it does not have a typedef name.

For example, here is a structure without a tag:

```
struct { ... }; /* structure with no tag */
```

Here is a structure with the tag point:

```
struct point { ... }; /* structure with tag */
```

Here is a `typedef` declaration that has no name:

```
typedef struct { ... }; /* typedef with no name */
```

Here is a `typedef` declaration with a name:

```
typedef struct { ... } complex; /* typedef with name */
```

190

`illegal usage of const attribute`

Cause

The C compiler issues this error if a `const` variable is used in the left-hand side of an assignment.

191

`illegal pointer conversion between const and non-const attribute`

Cause

The C compiler encountered an illegal pointer. Check to see if the program illegally passes a `const *` formal parameter to a reference parameter without the `const` attribute.

194

`invalid alias`

Cause

An external function name specified with an `_alias` attribute is one of these:

- Used in more than one declaration
- Applied to an object that is not a function
- Specified using incorrect syntax

195

`vararg function cannot have _extensible or _variable attribute`

Cause

The `vararg` function cannot be made into an extensible or variable function.

196

`illegal function attribute`

Cause

Function attributes are duplicated or conflict with each other. For example:

```
_extensible _extensible int foo(int x); /*_extensible specified twice */
_extensible _variable in for (int x); /*_extensible and _variable cannot
both be specified */
```

197

`illegal conversion to _extensible`

Cause

Only variable functions (functions with the `_variable` attribute) can be converted to extensible functions. The number specified in the `_extensible` parameter count argument must be in the range of 1 through 15.

198

```
illegal number of parameters count/size
```

Cause

There are either more than 252 parameters specified for a C regular function or more than 128 parameter words specified for variable or extensible function.

199

```
can be used in _extensible or _variable function only
```

Cause

The `_arg_present()` operator can be used with extensible functions only.

201

```
invalid structure alignment
```

Cause

An unknown `FIELDALIGN` pragma type name has been specified.

203

```
nn bits of filler inserted before this member for SHARED8 alignment
```

Cause

The specified number of filler bits was inserted before this member for `FIELDALIGN SHARED8` alignment.

204

```
nn bits of filler appended to the end of this structure for SHARED8 alignment
```

Cause

The specified number of filler bits was inserted after this member for `FIELDALIGN SHARED8` alignment.

205

```
tag name cannot be used with fieldalign pragma
```

Cause

Pragma `FIELDALIGN` is used with an enumeration tag. It can be used only with structure and union tags.

206

```
align type of tag name already set
```

Cause

You can only specify one `FIELDALIGN` setting for a tag.

207

```
address pointing at code space is taken
```

Cause

The address of an object that resides in code space is taken. Use of this address is valid only within the same code segment as the object.

208

```
string will be allocated within data space in this context, use pragma ENV
LIBRARY/LIBSPACE
```

Cause

A pointer to the code space has been initialized with the address of a string that has been allocated in the data space. Pragma `ENV LIBRARY` or `ENV LIBSPACE` must be used to ensure allocation of the string in the code space.

209

```
static variable not allowed under pragma ENV LIBRARY/LIBSPACE, convert it to
a constant
```

Cause

Under pragma `ENV LIBRARY` or `ENV LIBSPACE` , no relocatable data objects can be used. Therefore, no static variables can be declared. For read-only items, use type qualifier `const` .

210

```
constant pointer in code space not allowed
```

Cause

Pointers cannot reside in the code space.

211

```
_cspace must be accompanied by const
```

Cause

The `_cspace` qualifier must be specified in conjunction with the `const` qualifier.

212

```
initialization of pointer to code space not allowed use explicit assignment
or convert to array instead
```

Cause

A pointer cannot be initialized with an address that represents a location in the code space. You must assign the address at run time. To avoid this restriction, convert an array of string constants to a two-dimensional array of characters.

214

```
struct/union not allowed as parameter in _variable/_extensible function
```

Cause

Structures and unions cannot be specified as arguments to a function with the `_variable` or `_extensible` attributes.

215

```
making RTL call with argument pointing at code space under NOXMEM may lead to
undefined result, use CSADDR and NOINLINE
```

Cause

An address pointing to the code space is passed to a runtime routine under the small-memory model. This address is not valid in the code segment the runtime routine resides in. Pragmas `CSADDR` and `NOINLINE` must be used to ensure correct behavior.

216

`main function not allowed under ENV EMBEDDED/LIBSPACE`

Cause

The existence of the function `main()` is not meaningful under pragma `ENV EMBEDDED` or `ENV LIBRARY`.

217

`variable name and macro name conflict the macro name is installed`

Cause

A macro has been defined with the same name as an existing variable.

218

`no statement following a label`

Cause

A label has been specified immediately before the end of a compound statement.

222

`structure cannot contain members in code and data space at the same time`

Cause

All the members of a structure have to reside in the data space or the code space; mixing is not possible.

224

`number of bytes passed relies on previous declaration only`

Cause

The declarations for a pointer modified with the `_cspace` attribute are different. For example:

```
#pragma csaddr _cspace const struct S_tag { int i,j,k; } s = { 10, 20, 30 };
 /* Declared as 6 bytes */ func1 (_cspace const char * );
foo()
{
 _cpace const char *q; /* Declared as 1 byte */
q = (const char *) &s;
 foo1 ( q );
/* Only one byte copied and passed, */
} /* but points at a six-byte structure */
```

225

`local variable name` *`variable-name`* `same as function parameter`

Cause

A local variable and function parameter have the same name. For example:

```
int main(int argc, char *argv[]) { int argc; /* This line causes the warning */ }
```

226

`invalid value for enum literal` *`literal-name`*

Cause

Enumerated types must be compatible with the type `int` . Therefore, the value assigned to an enumerated type cannot exceed the range of integer for a given memory model. For example, an `int` in the large-memory model is 16 bits signed, so an enumerated value cannot exceed 32767.

228

`fatal error, compiler terminates`

Cause

The compiler issues this message if physical limitations prevent further compilation. The surrounding error messages give more details as to the cause of the failure.

229

`language specified for non-routine identifier, pragma function is ignored`

Cause

The identifier used in the pragma `FUNCTION` is not a known function. Ensure that the identifier has been declared.

230

`string` `is not a valid ANSI-compliant construct`

Cause

Use the type `double` instead of the type `long long` , which is not compliant with the ISO/ANSI C Standard.

233

`Unable to open input file` *`file-name`*

Cause

The compiler cannot open the input file. Check that the file is secured properly and the system is available.

234

`Unable to read input file` *`file-name`*

Cause

The compiler cannot read the input file. Check that the file is secured properly and the system is available.

235

`Unable to position input file`

Cause

The compiler cannot set the position for reading the input file.

236

`Unable to read source text`

Cause

The compiler cannot read the source text.

249

```
INTERNAL COMPILER ERROR
```

Cause

The compiler has detected an internal inconsistency. Contact your HP representative.

250

```
BINSERV or SYMSERV stopped
```

Cause

The compiler cannot communicate with the processes BINSERV or SYMSERV.

257

```
Compiler's heap space is totally full. Try breaking your program into smaller
modules.
```

Cause

The compiler cannot compile the program unless you break it into smaller modules.

261

```
Attempted division by constant zero
```

Cause

You cannot divide by a constant zero.

265

```
Please report this problem to HP Company. with a copy of your source code and
these stack trace.
```

Cause

An internal compiler error has occurred. Contact your HP representative and supply a copy of the source code and the stack trace, if possible.

282

```
Routine exceeds code size limit
```

Cause

The routine being compiled is too large to fit in the 32K words of code space. Break the routine into more than one routine.

289

`Parameter limit for extensible procedure is` *`number`* `words`

Cause

The extensible procedure has exceeded the upper bound of the number of words that can be passed to it.

294

```
Tag name is defined in fieldalign pragma but not used
```

Cause

The compiler requires that a tag specified in a `FIELDALIGN` pragma exist. Check to ensure that the correct tag was specified. Delete the `FIELDALIGN` pragma if it is not used.

295

```
0 bitfield not allowed under fieldalign shared2 and shared8
```

Cause

An unnamed bit field is not allowed in a structure or union tag with the `FIELDALIGN SHARED2` or `SHARED8` pragma.

296

```
Data block too long, truncated to 126 characters
```

Cause

The maximum length of a global variable name is 126 characters. The compiler truncates the name to 126 characters. Make sure that global variables have names that are unique within the first 126 characters.

299

```
Pragma columns different from previously specified, this pragma is ignored
```

Cause

All `COLUMNS` pragmas in a file must be the same.

303

```
Truncation of pointer size performed
```

Cause

The TNS C compiler has converted a 32-bit pointer to a 16-bit pointer, resulting in the loss of the segment portion of the address and leaving only the offset portion. This can occur in two instances:

- When the address of a location in extended memory is cast to a _ `near` pointer.
- When a non-C routine expecting a `_near` (16-bit) pointer argument is passed a 32-bit pointer.

For D4x releases, the default memory model is the large-memory model; for D3x releases, the default model is the small-memory model. If the location pointed to by the pointer is in extended memory, after converting to 16 bits, that location is lost.

For a memory location to be 16-bit addressable, it should be in the user data segment and not in the extended segment. Global/static scalars and all local variables are automatically in the user data segment. To place global/static aggregates in the user data segment, use the keyword `_lowmem` . Anything allocated on the heap is always in extended memory and hence is not 16-bit addressable.

307

```
Function definition overrides prior declaration of non-C routine
```

Cause

The user defined a C routine which was previously declared with a non-C language attribute. (The routine was declared as having been written in COBOL, FORTRAN, Pascal, or TAL.) The compiler disregards the language attribute and treats the function as a regular C function.

308

```
controlling expression for iteration must have scalar type
```

Cause

The controlling expression for iteration is not a scalar type.

309

`Passing unknown pragma directive`

Cause

The compiler cannot identify the pragma directive. Check the pragma to ensure that it is a valid directive.

310

`Invalid macro name`

Cause

An invalid macro name is defined.

311

`hexadecimal constant must have at least one digit`

Cause

A constant beginning with `"0X"` or `"0x"` has been defined without any following digits.

312

`escape sequence value must fit in unsigned char`

Cause

The escape sequence is too large to fit in an `unsigned char` value. Use a different data type.

313

`escape sequence value must fit in w_char`

Cause

The escape sequence is too large to fit in a `w_char` value. Use a different data type.

314

`Directive name is ignored on OSS environment`

Cause

A pragma that cannot be used in the OSS environment has been specified and ignored.

315

`Only the first sql (cppsource file) directive is accepted`

Cause

The compiler ignores subsequent `sql` (`cppsource` *file*) directives.

319

```
cpatheq directive has occurred more than once. Only the first definition has
effect.
```

Cause

Only the first occurrence of the `cpatheq` directive has effect.

320

```
mapinclude directive string text duplicated and ignored
```

Cause

A `mapinclude` directive was previously specified for this string.

321

```
variable sqlcode should be type short
```

Cause

The compiler requires the sqlcode variable to be declared type `short` . You have declared sqlcode to be a type other than type `short` .

322

```
Internal error: nested block level exceeds compiler limit; block ignored.
```

Cause

The C compiler supports a maximum of 255 block nesting levels. This message is generated when this limit is exceeded. Blocks nested at levels greater than 255 are ignored.

Run-Time Messages

This chapter presents the error messages that can occur during the execution of C and C++ programs. The Common Run-Time Environment (CRE) emits these messages. For more details on the error messages and the CRE, see the *Common Run‑Time Environment (CRE) Programmer’s Guide*.

Trap and Signal Messages

The TNS CRE reports the messages in this subsection if a trap occurs and your program has neither disabled traps nor enabled its own trap handler. The native CRE reports the messages in this subsection if a signal is raised and your program does not have its own signal handler for the signal that was raised. The CRE or run-time library terminates your program.

The CRE treats trap 4, arithmetic fault, as a program logic error, not a trap. If your program has disabled overflow traps, the TNS CRE returns control to your run-time library. The native CRE raises a SIGFPE signal instead. For additional detail, see the language-specific manuals for the routines in your program for additional detail. The *Guardian Programmer’s Guide* describes traps and signals.

1

```
Unknown trap
```

Message Explanation

Cause

The CRE trap processing function was called with an unknown trap number.

2

```
Illegal address reference
```

Message Explanation

Cause

An address was specified that was not within either the virtual code area or the virtual data area allocated to the process.

3

```
Instruction failure
```

Message Explanation

Cause

An attempt was made to:

- Execute a code word that is not an instruction.
- Execute a privileged instruction by a non privileged process.
- Reference an illegal extended address.

4

```
Arithmetic fault
```

Message Explanation

Cause

In the TNS environment, the overflow bit in the environment-register, ENV.<10>, was set to 1. In the native environment, a SIGFPE was raised. In either environment, the fault occurs for one of these reasons:

- The result of a signed arithmetic operation could not be represented with the number of bits available for the particular data type.
- A division operation was attempted with a zero divisor.

5

```
Stack overflow
```

Message Explanation

Cause

A stack overflow fault occurs if:

- An attempt was made to execute a procedure or subprocedure whose local or sublocal data area extends into the upper 32K of the user data area.
- There was not enough remaining virtual data space for an system procedure to execute.
- The native environment exceeded the maximum stack space available.

The amount of virtual data space available is G[0] through G[32767].

System procedures require approximately 350 words of user-data stack space to execute.

6

```
Process loop-timer timeout
```

Message Explanation

Cause

The new time limit specified in the latest call to SETLOOPTIMER has expired.

7

```
Memory manager read error
```

Message Explanation

Cause

An unrecoverable, read error occurred while the program was trying to bring in a page from virtual memory.

8

```
Not enough physical memory
```

Message Explanation

Cause

This fault occurs for one of these reasons:

- A page fault occurred, but there were no physical memory pages available for overlay.
- Disk space could not be allocated while the program is using extensible segments.

9

```
Uncorrectable memory error
```

Message Explanation

Cause

An uncorrectable memory error occurred.

10

```
Interface limit exceeded
```

Message Explanation

Cause

The operating system reported a trap 5.

CRE Service Function Messages

The CRE writes the messages in this subsection if an error occurs during its own processing or if it receives a request from a run-time library to report a specific message.

11

```
Corrupted environment
```

Message Explanation

Cause

CRE or run-time library data is invalid.

Effect

The CRE invokes PROCESS_STOP_, specifying the ABEND variant and the text “Corrupted environment.”

Recovery

In the TNS environment, the program might have written data in the upper 32K words of the user data segment. The upper 32K words are reserved for TNS CRE and run-time library data.

In the native environment, the run-time environment has been corrupted. You may have written data over run-time data structures.

Check the program’s logic. Use a symbolic debugger to help isolate the problem or consult your system administrator.

12

```
Logic error
```

Message Explanation

Cause

The CRE or a run-time library detected a logic error within its own domain. For example, although each data item it is using is valid, the values of the data items are mutually inconsistent.

Effect

The CRE invokes PROCESS_STOP_, specifying the ABEND variant and the text “Logic error.”

Recovery

In the TNS environment, the program might have written data in the upper 32K words of the user data segment. The upper 32K words are reserved for TNS CRE and run-time library data.

In the native environment, the run-time environment has been corrupted. You may have written data over run-time data structures.

Check the program’s logic. Use a symbolic debugger to help isolate the problem or consult your system administrator.

13

```
MCB pointer corrupt
```

Message Explanation

Cause

The pointer at location G[0] of the program’s user data segment to its primary data structure—the Master Control Block (MCB)—does not point to the MCB. Both the CRE and run-time libraries can report this error.

In the native environment, this error can occur if the MCB run-time data structure has been corrupted.

Effect

The CRE attempts to restore the pointer at G[0] and to write a message to the standard log file. However, because its environment might be corrupted, the CRE might not be able to log a message. In that case, it calls PROCESS_STOP_, specifying the ABEND variant, and the text “Corrupted Environment”.

Recovery

Check the program’s logic to see if it overwrote the MCB pointer at G[0]. Use a symbolic debugger to help isolate the problem.

14

```
Premature takeover
```

Message Explanation

Cause

The CRE backup process received an operating system message that it had become the primary process but it had not yet received all of its initial checkpoint information from its predecessor primary process.

Effect

The CRE invokes PROCESS_STOP_, specifying the ABEND variant and the text “Premature takeover.”

Recovery

If the takeover occurred because of faulty program logic, correct the program’s logic. If the takeover occurred for other reasons, such as a hardware failure, you might want to rerun the program. Do not rerun the program if doing so will duplicate operations already performed, such as updating a database a second time.

15

```
Checkpoint list inconsistent
```

Message Explanation

Cause

A list of checkpoint item descriptors that the CRE maintains for fault-tolerant processes was invalid.

Effect

The CRE terminates the program.

Recovery

The list of items to checkpoint is maintained in the program's address space. Check the program's logic. The program might have overwritten the checkpoint list. Use a symbolic debugger to help isolate the problem.

16

```
Checkpoint list exhausted
```

Message Explanation

Cause

The CRE did not have enough room to store all the checkpoint information required by the program.

Effect

Program behavior is language and application dependent.

Recovery

Increase the checkpoint list object's size. For the routine that allocates your checkpoint list, see the language manual.

17

```
Cannot obtain control space
```

Message Explanation

Cause

The CRE or a run-time library could not obtain heap space for all of its data.

Effect

If the request came from the CRE, it terminates the program. Otherwise, program behavior is language and application dependent.

Recovery

You might be able to increase the amount of control space available to your program by reducing the number of files your program has open at the same time or by decreasing the size of buffers allocated to open files.

18

```
Extended Stack Overflow
```

Message Explanation

Cause

A module could not obtain sufficient extended stack space for its local data.

Effect

Program behavior is language and application dependent.

Recovery

Increase the extended stack's size. See the language manual for the routine that caused the extended stack overflow for details on increasing the size of the extended stack.

19

`Incompatible runtime libraries - `*`Library1`*` vs. `*`Library2`*

Message Explanation

Library1

Library2

The names of the incompatible libraries

Cause

Libraries that are not compatible with each other are linked into the application.

Effect

The CRE terminates the program.

Recovery

Identify which library is the correct library and rebuild the application such that the incompatible library is excluded.

20

```
Cannot utilize filename
```

Message Explanation

Cause

A string, expected to be a valid file name, could not be manipulated as a Guardian external file name.

Effect

If the file name came from the CRE, the program is terminated. Otherwise, program behavior is language and application dependent.

Recovery

Check that the file names in the program are valid Guardian file names.

21

`Cannot read initialization messages ( `*`error`*` )`

Message Explanation

Cause

During program initialization, the CRE could not read all the messages (start-up message, PARAM message, ASSIGN messages, and so forth) it expected from the file system. *error* is the file-system error number the CRE received when it couldn't read an initialization message.

Effect

The CRE terminates the program.

Recovery

Consult your system administrator.

22

```
Cannot obtain program filename
```

Message Explanation

Cause

The CRE could not obtain the name of the program file from the operating system.

Effect

The CRE terminates the program.

Recovery

Consult your system administrator.

23

`Cannot determine filename (` *`error`* `)` *`program_name.logical_name`*

Message Explanation

Cause

The CRE could not determine the physical file name associated with *program_name.logical_name*.

Effect

The CRE terminates the program.

Recovery

Correct the *program_name.logical_name* and rerun your program. For general information on ASSIGN commands, see the *TACL Reference Manual*. For more details on using ASSIGNs, see the reference manual for your program's 'main' routine.

24

`Conflict in application of ASSIGN` *`program_name.logical_name`*

Message Explanation

Cause

ASSIGN values in your TACL environment conflict with each other. For example:

```
ASSIGN   A, $B1.C.D
ASSIGN *.A, $B2.C.D
```

The first ASSIGN specifies that the logical name A can appear in no more than one program file. The second assign specifies that the name A can appear in an arbitrary number of program files. The CRE cannot determine whether to use the file C.D on volume $B1 or on volume $B2.

Effect

The CRE terminates the program.

Recovery

Correct the ASSIGNs in your TACL environment. For more details on using ASSIGNs, see the *TACL Reference Manual*.

25

`Ambiguity in application of ASSIGN` *`logical_name`*

Message Explanation

Cause

Your TACL environment specifies an ASSIGN such as:

```
ASSIGN   A, $B1.C.D
```

but the program contains more than one logical file named A.

Effect

The CRE terminates the program.

Recovery

Correct the ASSIGNs in your TACL environment. For more details on using ASSIGNs, see the *TACL Reference Manual*.

26

`Invalid PARAM value text ( error ) PARAM name 'value'`

Message Explanation

Cause

A PARAM specifies a value that is not defined by the CRE. For example, the value for a DEBUG PARAM must be either ON or OFF:

```
PARAM DEBUG [ ON ]
            [ OFF ]
```

The CRE reports this error if a DEBUG PARAM has a value other than ON or OFF. *error*, if present, is a file-system error.

Effect

The CRE terminates the program.

Recovery

Modify the PARAM text and rerun your program. For more details on using ASSIGNs, see the *TACL Reference Manual*.

27

`Ambiguity in application of PARAM PARAM name 'value'`

Message Explanation

Cause

A PARAM specifies a value that is ambiguous in the current context. For example, the PARAM specification:

```
PARAM PRINTER-CONTROL A
```

is ambiguous if the program contains more than one logical file named A.

Effect

The CRE terminates the program.

Recovery

Correct the PARAM in your TACL environment. For more details on using ASSIGNs, see the *TACL Reference Manual*.

28

`Missing language run-time library --` *`language`*

Message Explanation

Cause

The run-time library for a module that is written in *language* is not available to the program.

Effect

The CRE terminates the program.

Recovery

Consult your system administrator.

29

`Program incompatible with run-time library -- [`*`language`*` | `*`library`*`]`

Message Explanation

Cause

One of these conditions occurred:

- The *language* compiler used features that are not supported by the *language* run-time library that the program used.
- The application used a function or feature that was not compatible with the *library* to which it was linked.

Effect

The CRE terminates the program.

Recovery

Use a compiler and run-time library that are compatible. You might need to consult your system administrator.

Ensure that the application is linked to the correct libraries, in the correct order, for the functions or features used. For example, if you are using the version of the `getgrent_r()` function that uses three parameters, ensure that you specify linking to the Standard POSIX Threads library (ZSPTDLL) before you specify linking to the public library ZSECDLL.

Heap-Management Messages

The CRE or run-time libraries report the messages in this subsection if they detect an error while accessing the heap.

30

`Unknown heap error`

Message Explanation

Cause

A heap-management routine was called with an invalid error number.

Effect

The CRE terminates the program.

Recovery

Refer to the reference manual that corresponds to the language in which the routine that requests heap space is written. You might need to consult your system administrator.

31

```
Cannot obtain data space
```

Message Explanation

Cause

Your program, or the run-time library for one of the modules in your program, requested more space on the heap than is currently available.

Effect

Program behavior is language and application dependent.

Recovery

Refer to the reference manual that corresponds to the language in which the routine that requests heap space is written. You might need to consult your system administrator.

32

```
Invalid heap or heap control block Process heap size is 0
```

Message Explanation

Cause

The CRE or a run-time library found invalid data in the user heap or in the heap control block.

Your program might be writing information over the heap or heap control block. An invalid pointer or indexing operation could cause this error.

“Process heap size is 0” appears when your program or a run-time library requested space, but the process has no user heap.

Effect

Program behavior is language and application dependent.

Recovery

In the TNS environment, the program might have written data in the upper 32K words of the user data segment or in the extended segment. The upper 32K words of the user data area are reserved for CRE and run-time library data. In a small-memory model, the heap is allocated in the lower 32K words of the user data segment. In a large memory model, the heap is allocated in an extended memory segment. Check the program’s logic. Use a symbolic debugger to help isolate the problem or consult your system administrator.

If this error occurs in the native environment, use a symbolic debugger to help isolate the problem.

In the case of “Process heap size is 0,” specify the existence of a user heap when building your program.

33

```
Released space address is 0
```

Message Explanation

Cause

The address of a block of memory to return to the heap was zero.

Effect

Program behavior is language and application dependent.

Recovery

Correct the program to pass the correct address.

34

```
Released space not allocated
```

Message Explanation

Cause

The address of a block of memory to return to the heap did not point to a block allocated from the heap.

Effect

Program behavior is language and application dependent.

Recovery

Correct the program to return the correct pointer value.

35

```
Heap corrupted ( 32-bit_octal_address )
```

Message Explanation

Cause

The CRE or a run-time library found invalid information at location *32-bit_octal_address* of the heap.

Effect

Program behavior is language and application dependent.

Recovery

The program might have written data over the heap. In a small-memory model, the heap is allocated in the lower 32K words of the user data segment, just below the run-time stack. In a large memory model, the heap is allocated in an extended memory segment. Check the program's logic. Use a symbolic debugger to help isolate the problem or consult your system administrator.

If this error occurs in the native environment, check the program's logic. Use a symbolic debugger to help isolate the problem or consult your system administrator.

36

```
Invalid operation for heap
```

Message Explanation

Cause

The request you made is not compatible with the heap that you referenced.

Effect

Program behavior is language and application dependent.

Recovery

Consult your system administrator.

37

```
Mark address or space corrupt
```

Message Explanation

Cause

An address passed as a heap marker does not point to a mark.

Effect

Program behavior is language and application dependent.

Recovery

In the TNS environment, ensure that the program passed the correct address of a mark. If it did, the heap might be corrupted. Check the program’s logic. CRE and run-time library data is stored in the upper 32K words of the user data segment and in the primary extended data segment. The program might have overwritten CRE or run-time library data. Use a symbolic debugger to help isolate the problem. You might need to consult your system administrator.

If this error occurs in the native environment, check the program’s logic. Use a symbolic debugger to help isolate the problem or consult your system administrator.

39

`C signal raised --` *`signal`*

Message Explanation

Cause

The `raise()` function is executed in a TNS C program. *signal* identifies the signal (for example. SIGTERM).

Effect

The CRE terminates the program.

Recovery

Modify the application so that it does not invoke `raise()`.

Function Parameter Message

Run-time libraries report the message in this subsection if an error is detected in a parameter passed to a function.

40

```
Invalid function parameter
```

Message Explanation

Cause

A function detected a problem with its parameters.

Effect

Program behavior is language and application dependent.

Recovery

Correct the parameter you are passing.

Math Function Messages

Run-time libraries report the messages in this subsection if an error is detected in a math function.

41

```
Range fault
```

Message Explanation

Cause

An arithmetic overflow or underflow occurred while evaluating an arithmetic function.

Effect

Program behavior is language and application dependent.

Recovery

Modify the program to pass values to the arithmetic functions that do not cause overflow.

42

```
Arccos domain fault
```

Message Explanation

Cause

The parameter passed to the arccos function was not in the range:

`-1.0 <` *`parameter`* `< 1.0`

Effect

Program behavior is language and application dependent.

Recovery

Modify the program to pass a valid value to the arccos function.

43

```
Arcsin domain fault
```

Message Explanation

Cause

The parameter passed to the arcsin function was not in the range:

`-1.0 <` *`parameter`* `< 1.0`

Effect

Program behavior is language and application dependent.

Recovery

Modify the program to pass a valid value to the arcsin function.

44

```
Arctan domain fault
```

Message Explanation

Cause

Both of the parameters to an arctan2 function were zero. At least one of the parameters must be nonzero.

Effect

Program behavior is language and application dependent.

Recovery

Modify the program to pass the correct value to the arctan2 function.

46

```
Logarithm function domain fault
```

Message Explanation

Cause

The parameter passed to a logarithm function was less than or equal to zero. The parameter to a logarithm function must be greater than zero.

Effect

Program behavior is language and application dependent.

Recovery

Modify the program to pass a valid value to the logarithm function.

47

```
Modulo function domain fault
```

Message Explanation

Cause

The value of the second parameter to a modulo function was zero. The second parameter to a modulo function must be nonzero.

Effect

Program behavior is language and application dependent.

Recovery

Modify the program to pass a nonzero value to the modulo function.

48

```
Exponentiation domain fault
```

Message Explanation

Cause

Parameters to a Power function were not acceptable. Given the expression

```
xy
```

these parameter combinations produce this message:

```
x = 0 and y < 0
x < 0 and y is not an integral value
```

Effect

Program behavior is language and application dependent.

Recovery

Modify the program to pass values that do not violate the above combinations.

49

```
Square root domain fault
```

Message Explanation

Cause

The parameter to a square root function was a negative number. The parameter must be greater than or equal to zero.

Effect

Program behavior is language and application dependent.

Recovery

Modify the program to pass a nonnegative value to the square root function.

Function Parameter Messages

The CRE or run-time libraries report the messages in this subsection if there is a problem with the parameters passed to a function.

55

```
Missing or invalid parameter
```

Message Explanation

Cause

A required parameter is missing or too many parameters were passed.

Effect

Program behavior depends on the function that was called and the language in which it is written.

Recovery

Correct the program to pass a valid parameter.

56

```
Invalid parameter value
```

Message Explanation

Cause

The value passed as a procedure parameter was invalid.

Effect

Program behavior depends on the function that was called and the language in which it is written.

Recovery

Correct the program to pass a valid parameter value.

57

```
Parameter value not accepted
```

Message Explanation

Cause

The value passed as a procedure parameter is not acceptable in the context in which it is passed. For example, the number of bytes in a write request is greater than the number of bytes per record in the file.

Effect

Program behavior depends on the function that was called and the language in which it is written.

Recovery

Correct the program to pass a valid parameter.

Input/Output Messages

The CRE or run-time libraries report the messages in this subsection if an error occurs when calling an I/O function.

59

`Standard input file error ( error ) Unable to open filename`

Message Explanation

Cause

The file system reported an error when a routine tried to access the standard input file. *error* is a file-system error number.

“`Unable to open` *filename*” appears when the C run-time library cannot open the standard input file during program initialization. *filename* shows the name for which the open operation failed.

Effect

The CRE can report this error when it closes your input file. All other instances are language and application dependent.

Recovery

If the error was caused by a read request from your program, correct your program. You might need to ensure that your program handles conditions that are beyond your control such as losing a path to the device. Also refer to error handling in this manual and in the language manual for the routine in your program that detected the error.

If the error was caused by a read request from the CRE, consult your system administrator.

If the error was caused during program initialization, specify an acceptable input file when executing your program.

60

`Standard output file error ( error ) Unable to open filename`

Message Explanation

Cause

The file system reported an error when the CRE called a file system procedure to access standard output. *error* is the file-system error number.

“`Unable to open` *filename*” appears when the C run-time library cannot open the standard output file during program initialization. *filename* shows the name for which the open operation failed.

Effect

The CRE can report this error when it closes your output file. All other instances are language and application dependent.

Recovery

If the error was caused by a write request from your program, correct your program. You might need to ensure that your program handles conditions that are beyond your control such as losing a path to the device. Also refer to error handling in this manual and in the language manual for the routine in your program that detected the error.

If the error was caused by a write request from the CRE, consult your system administrator.

If the error was caused during program initialization, specify an acceptable output file when executing your program.

61

`Standard log file error (` *`error`* `) Unable to open` *`filename`*

Message Explanation

Cause

The file system reported an error when the CRE called a file system procedure to access the standard log file. *error* is the file system error number.

"`Unable to open` *filename*" appears when the C run-time library cannot open the standard log file during program initialization. *filename* shows the name for which the open operation failed.

Effect

The CRE terminates your program.

Recovery

Consult your system administrator.

If the error was caused during program initialization, specify an acceptable log file when executing your program.

62

`Invalid GUARDIAN file number`

Message Explanation

Cause

A value that is expected to be a Guardian file number is not the number of an open file.

Effect

Program behavior is language and application dependent.

Recovery

Consult your system administrator.

63

`Undefined shared file`

Message Explanation

Cause

A parameter was not the number of a shared (standard) file where one was expected.

Effect

Program behavior is language and application dependent.

Recovery

Consult your system administrator.

64

```
File not open
```

Message Explanation

Cause

A request to open a file failed because the file device is not supported.

Effect

Program behavior is language and application dependent.

Recovery

Consult your system administrator.

65

```
Invalid attribute value
```

Message Explanation

Cause

A parameter to an open operation was not a meaningful value. For example, the CRE_File_Open_*sync_receive_depth* parameter must be a nonnegative number. This message might be reported if the *sync_receive_depth* parameter is negative.

Effect

Program behavior is language and application dependent.

Recovery

Consult your system administrator.

66

```
Unsupported file device
```

Message Explanation

Cause

The CRE received a request to access a device that it does not support.

Effect

Program behavior is language and application dependent.

Recovery

Consult your system administrator.

67

```
Access mode not accepted
```

Message Explanation

Cause

The value of the *access* parameter to an open operation was not valid in the context in which it was used. For example, it is invalid to open a spool file for input.

Effect

Program behavior is language and application dependent.

Recovery

Consult your system administrator.

68

```
Nowait value not accepted
```

Message Explanation

Cause

The value of the *no_wait* parameter to an open operation was not valid in the context in which it was used. For example, it is invalid to specify a nonzero value for *no_wait* for a device that does not support nowait operations.

Effect

Program behavior is language and application dependent.

Recovery

Consult your system administrator.

69

```
Syncdepth not accepted
```

Message Explanation

Cause

The value of the *sync_receive_depth* parameter to an open operation was not valid in the context in which it was used. For example, it is not valid to specify a *sync_receive_depth* greater than one for a standard file.

Effect

Program behavior is language and application dependent.

Recovery

Consult your system administrator.

70

```
Options not accepted
```

Message Explanation

Cause

The value of an open operation *options* parameter was not valid in the context in which it was used.

Effect

Program behavior is language and application dependent.

Recovery

Consult your system administrator.

71

```
Inconsistent attribute value
```

Message Explanation

Cause

A routine requested a connection to a standard file that was already open, and the attributes of the new open request conflict with the attributes specified when the file was first opened.

Effect

Program behavior is language and application dependent.

Recovery

If your program supplied the attribute values, correct and rerun your program. Otherwise, consult your system administrator.

75

```
Cannot obtain buffer space
```

Message Explanation

Cause

A routine was not able to obtain buffer space.

Effect

Program behavior is language and application dependent.

Recovery

Program recovery is language and application dependent.

76

`Invalid external filename (` *`error`* `)`

Message Explanation

Cause

A value that was expected to be a Guardian external file name is not in the correct format.

Effect

Program behavior is language and application dependent.

Recovery

If you supplied an invalid file name, correct the file name and rerun your program. Otherwise, consult your system administrator.

77

`EDITREADINIT failed (` *`error`* `)`

Message Explanation

Cause

A call to EDITREADINIT failed. *error*, if present, gives the reason for the failure. Possible values of *error* are:

Effect

Program behavior is language and application dependent. For more details, see the *Guardian Procedure Calls Reference Manual*.

Recovery

Recovery is language and application dependent.

78

```
EDITREAD failed ( error )
```

Message Explanation

Cause

A call to EDITREAD failed. *error*, if present, gives the reason for the failure.

Effect

Program behavior is language and application dependent.

Recovery

For more details, see the *Guardian Procedure Calls Reference Manual*.

79

```
OpenEdit failed ( error )
```

Message Explanation

Cause

A call to OPENEDIT_ failed. *error*, if present, is the error returned by OPENEDIT_. A negative number is a format error. A positive number is a file-system error number.

Effect

Program behavior is language and application dependent.

Recovery

For more details, see the *Guardian Procedure Calls Reference Manual*.

80

```
Spooler initialization failed ( error )
```

Message Explanation

Cause

An initialization operation to a spooler collector failed. *error*, if present, is the file-system error number returned by the SPOOLSTART system procedure.

Effect

Program behavior is language and application dependent.

Recovery

For more details, see the *Spooler Programmer’s Guide*.

81

```
End of file
```

Message Explanation

Cause

A routine detected an end-of-file condition.

Effect

Program behavior is language and application dependent.

Recovery

Correct your program to allow for an end-of-file condition or ensure that your program can determine when all the data has been read.

82

```
Guardian I/O error nnn
```

Message Explanation

Cause

An operating system routine returned file-system error *nnn*. This error is usually reported as a result of an event that is beyond control of your program such as a path or system is not available.

Effect

Program behavior is language and application dependent.

Recovery

Consult your system administrator.

83

```
Operation incompatible with file type or status (error)
```

Message Explanation

Cause

The program attempted an operation on a file whose type or current status is unsuitable for execution of that operation. For example, a COBOL program calls a file manipulation utility for a file that is using Fast I/O. *error*, if present, is as file-system error number.

Effect

Program behavior is language and application dependent.

Recovery

Change the program so that it does not attempt the operation on an unsuitable file.

90

```
Open conflicts with open by other language
```

Message Explanation

Cause

Routines written in two different languages—for example, COBOL and FORTRAN—attempted to open a connection to a file using the same logical name. This error is not reported for standard files.

Effect

Program behavior is language and application dependent.

Recovery

Modify your program to use different logical names or coordinate logical names between the two routines so that they do not open the same logical file at the same time.

91

```
Spooler job already started
```

Message Explanation

Cause

A FORTRAN routine attempted to open a spooler but the spooler was already open with attributes that conflict with those in the current open. This error is reported only for an open to standard output and only if one or more of these are true:

- The spooling levels of the two opens are different.
- The new open specifies any level-2 arguments.

Effect

If the rejected request was initiated from a FORTRAN I/O statement that includes either an IOSTAT or ERR parameter, control is returned to the FORTRAN routine. Otherwise, the FORTRAN run-time library terminates the program.

Recovery

Coordinate how routines in your program use standard output.

Environment Messages

275

`Ambiguity in application of` *`errno`*

Message Explanation

Cause

errno is defined in the user's object file. The native CRE reports this error if *errno* has been defined in the object file, and it is not the same *errno* defined by the native CRE shared run-time library instance data item *errno*.

Effect

The run-time library terminates your program.

Recovery

Make sure the only defined *errno* in a program is the one defined in the native CRE shared run-time library.

276

```
environAmbiguity in application of
```

Message Explanation

Cause

`environ` is defined in the user's object file. The native CRE reports this error if `environ` has been defined in the object file, and it is not the same `environ` defined by the native CRE shared run-time library instance data item `environ`.

Effect

The run-time library terminates your program.

Recovery

Make sure the only defined `environ` in a program is the one defined in the native CRE shared run-time library.

Handling TNS Data Alignment

Programs compiled with the TNS instruction set must follow the even-byte data alignment rules of TNS compilers. Certain violations of these programming rules might lead to run-time errors. This section describes these violations and explains how to avoid them and how to diagnose them at run time.

On TNS systems, a word is 16 bits. The TNS instruction set includes data access instructions that expect 32-bit byte addresses that must be even-byte aligned (that is, aligned 0 modulo 2) for correct operation. In TNS mode and Accelerator mode, addresses that are odd-byte aligned (that is, aligned 1 modulo 2) are called misaligned. TNS processors consistently “round down” misaligned addresses (that is, they ignore the low-order bit).

TNS/R, TNS/E, and TNS/X processors handle misaligned addresses of TNS programs inconsistently, rounding down some but not others and behaving differently in TNS mode and accelerated mode. Compilers cannot catch misaligned addresses that are computed at run time.

The behavior of TNS programs with misaligned addresses on TNS/R, TNS/E, or TNS/X processors is almost impossible to predict. If you are not sure that your program has only aligned addresses, you can use the tracing facility to detect whether programs are using misaligned pointers, and if so, where. You should then change the programs to avoid misalignment.

The round-downs occur only in TNS-compiled C and C++ programs, native C or C++ programs. In native mode, misalignments can slow down a program, but they cannot cause errors.

The data misalignment issue might affect programs that use these HPE products. If you use these products, see the appropriate manual or addendum:

Product	Number	Manual or Addendum
TNS C	T9255	This manual
TNS C++	T9541	This manual
G-series TNS c89	T8629	This manual
TNS/E c89	T8164	This manual
TNS/E CCOMP	T9577	This manual
TNS/E CPPCOMP	T9225	This manual
TNS/R c89	T8164	This manual
TNS/R NMC	T9577	This manual
TNS/R NMCPLUS	T9225	This manual
Accelerator	T9276	*Accelerator Manual*
TNS COBOL85 and OSS `cobol`	T9257	*COBOL Manual for TNS and TNS/R*
TNS/E EpTAL	T9248	*pTAL Reference Manual*

Table Continued

Product	Number	Manual or Addendum
TNS/R pTAL	T9248	*pTAL Reference Manual*
TNS TAL	T9250	*TAL Programmer's Guide Data Alignment Addendum*

Misalignment Tracing Facility

The misalignment tracing facility is enabled or disabled on a system-wide basis (that is, for all processors in the node). By default, it is enabled (set to ON). It can be disabled (set to OFF) only by the persons who configure the system, using the Subsystem Control Facility (SCF) attribute MISALIGNLOG. Instructions are in the *SCF Reference Manual for the Kernel Subsystem*.

NOTE: HPE recommends that the SCF MISALIGNLOG attribute be left ON (its default setting) so that any process that is subject to rounding of misaligned addresses will generate log entries, facilitating diagnosis and repair of the code. Only if the volume of misalignment events becomes burdensome should this attribute be turned OFF.

The tracing facility traces exceptions that would be rounded down on a NonStop system running any release version update (RVU) earlier than G06.17.

The tracing facility does not count and trace every misaligned address, only those that cause round-down exceptions. Other accesses that use misaligned addresses without rounding them down do not cause exceptions and are not counted or traced. Also, only a periodic sample of the counted exceptions are traced by means of their own EMS event messages.

While a process runs, the tracing facility:

- Counts the number of misaligned-address exceptions that the process causes (the exception count)
- Records the program address and code-file name of the instruction that causes the first misaligned-address exception

Sampling

Because a process can run for a long time, the tracing facility samples the process (that is, checks its exception data) approximately once an hour. If the process recorded an exception since the previous sample, the tracing facility records an entry in the Event Management Service (EMS) log. If the process ends, and any exception has occurred since the last sample, the operating system produces a final EMS event.

The EMS event includes:

- The process's exception count
- Details about one misaligned-address exception, including the program address, data address, and relevant code-file names

Sampling is short and infrequent enough to avoid flooding the EMS log, even for a continuous process with many misaligned-address exceptions. One sample logs a maximum of 100 events, and at most one event is logged for any process.

If misaligned-address exceptions occur in different periods of a process, the operating system produces multiple EMS events for the same process, and these EMS events might have different program addresses.

For more details about EMS events or the EMS log, see the *EMS Manual*.

Misalignment Handling

Misalignment handling is determined by these SCF attributes, which are set system-wide (that is, for all processors in the node) during system configuration:

- MISALIGNLOG

  MISALIGNLOG enables or disables the tracing facility (see **Misalignment Tracing Facility** on page 498).

- TNSMISALIGN

  TNSMISALIGN applies to all programs running in TNS mode or accelerated mode.

- NATIVEATOMICMISALIGN

  NATIVEATOMICMISALIGN applies to calls to pTAL atomic functions in programs running in native mode. Non-atomic accesses in native mode are always NOROUND (For the definition of NOROUND, see **TNS Misalignment Handling Methods**).

TNS Misalignment Handling Methods lists and describes the possible settings for TNSMISALIGN. Each setting represents a different misalignment handling method. For more details about TNSMISALIGN, see the *SCF Reference Manual for the Kernel Subsystem*.

Table 66: TNS Misalignment Handling Methods

Method	Description
ROUND (default)*	After rounding down a misaligned address, the system proceeds to access the address, as in G06.16 and earlier RVUs.
FAIL	Instead of rounding down a misaligned address and proceeding to access the target, the operating system considers the instruction to have failed.For a Guardian process, this failure generates an Instruction Failure trap (trap #1). By default, this trap causes the process to go into the debugger, but the program can specify other behavior (for example, process termination or calling a specified trap-handling procedure). For information about trap handling, see the *Guardian Programmer’s Guide*. For an OSS process, this failure generates a SIGILL signal (signal #4). By default, this signal causes process termination, but the program can specify other behavior (for example, entering the debugger or calling a specified signal-handler procedure). The signal cannot be ignored. For information about signal handling, see the explanation of the `sigaction()` function in the *Open System Services System Calls Reference Manual*.
NOROUND	The system uses the operand’s given odd address (not rounded down) to complete the operation. If the operation is an atomic operation, atomicity is no longer guaranteed.

* Use this method on production systems to avoid changing the semantics of old TNS programs. FAIL could cause possibly fatal Instruction Failure traps in faulty TNS programs. NOROUND might change the semantics of some faulty programs.

- The method that you choose does not apply to every misaligned address, only to those that would have been rounded down in RVUs prior to G06.17.
- ROUND and NOROUND misalignment handling are both intended as temporary solutions, not as a substitute for changing your program to ensure that it has only aligned addresses.
- Programs that depend on NOROUND misalignment handling cannot be safely migrated to all present and future NonStop OS platforms or to systems configured with ROUND or FAIL misalignment handling.
- Programs that depend on ROUND misalignment handling cannot be safely migrated "as is" to future NonStop platforms or to systems configured with NOROUND or FAIL misalignment handling.

C/C++ Misalignment Examples

In native mode, the targets of C and C++ pointers are usually aligned for best performance, but full alignment is not required. Misaligned addresses might slow down a native program, but the misaligned addresses are never "rounded down" in native mode and do not cause the program to terminate abnormally.

In TNS mode and accelerated mode, the targets of C and C++ pointers (except pointers to `char` items) must be aligned on 2-byte memory boundaries for correct operation. The results of odd-byte addresses depend on the specific NonStop server and the system configuration, but they might include erratic "rounding down" and abnormal program termination.

In a TNS-compiled C or C++ program, some actions that can cause misaligned addresses are:

1. Using a null or uninitialized pointer.

When a structure pointer is null or uninitialized, the structure's fields are random bits, which could give a random misaligned address if a pointer field is dereferenced. Examples:

a. Dereferencing a pointer-valued field of a structure when the structure pointer is null

b. Checking for a null pointer after some but not all pointer references, instead of before all pointer references (see **C/C++ Null Pointer Check (Item 1b)** on page 502).

c. Dereferencing a pointer-valued member of a union when that alternative is not active and initialized

2. Using syntactically correct but semantically incorrect and nonportable casts of these types:

- To an integer pointer
- To a structure pointer
- From a `char *` pointer
- From a `void *` pointer
- From an integer expression (For information about writing portable programs, see **Writing Portable Programs** on page 48.)

  These casts work only in TNS programs when the converted address is correctly aligned for integer or structure objects (that is, when it is an even-byte address).

  These are correctly aligned (that is, they have even-byte addresses):

- Compiler-allocated objects

- Heap objects

  These might be misaligned (that is, they might have odd-byte addresses):

- Declared items
  - char strings
  - Elements of char arrays
  - char fields of structures
- Calculated items
  - char subscripts
  - Incremented char or void pointers
  - Casts of odd-valued integer expressions into a pointer types

  See:

- **C/C++ Invalid Cast From char to Integer Pointer (Item 2, Item 10)** on page 503
- **C/C++ Invalid Cast From char to Integer Pointer (Item 2, Item 10)** on page 503
- **C/C++ Check-Sum Function (Item 2, Item 9)** on page 504

3. Using a pointer union without direct assignment

Union of a pointer with other pointers or with integers is safe when the values of the unions are mutually exclusive in time or when an equivalent explicit assignment of the address value would be correct at run time. See **C/C++ Pointer Union (Item 3, Item 10)** on page 503.

4. Calling an undeclared or misdeclared external function

Actual parameter values are implicitly assigned to formal parameters. If formal parameters are described incorrectly, these implicit assignments are equivalent to unchecked type casts.

5. Using an explicit number for the offset of a structure field or for the size of a structure

This practice can cause misaligned addresses by overlooking the implicit filler bytes that the TNS compiler adds to structures:

- Within structures, to ensure that every noncharacter field begins at an even-byte offset from the beginning of the structure
- At the end of any structure that contains some noncharacter fields and has an odd number of bytes, to give it an even number of bytes

See **C/C++ Structure With Implicit Filler Bytes (Item 5)** on page 504.

To prevent this problem, use an `offsetof()` macro to get the offset of a structure field and the `sizeof` instruction to get the size of a structure.

6. Customizing the heap allocation method

If a TNS program implements its own customized heap allocation method, it must ensure that all objects except `char` objects are aligned and allocated on even-byte boundaries.

7. Appending a sequence of objects into a string buffer or `char` array

If a TNS program appends a sequence of objects into a string buffer or a `char` array, it must ensure that nonstring objects are aligned and allocated on even-byte boundaries.

8. Addressing an external data item that contains misaligned parts (as `short` or larger items)

If a TNS program uses external data items (files or structures) that have misaligned parts (such as those on computer systems that have no data alignment requirements), it must declare and directly access them as `char` arrays rather than as `short` or larger items. See **C/C++ External Structure With Misaligned Parts (Item 8)** on page 504.

9. Using string check-sum or hashing functions that use 16-bit or 32-bit ADD or XOR operations

Some TNS programs use check-sum or hashing functions that use a series of 16-bit or 32-bit ADD or XOR operations to reduce an arbitrary length byte string to 16 or 32 bits. If the string begins on an even-byte boundary, this is often coded as an array of `short` or `int` elements overlaying the byte string, but if the string can begin at an arbitrary address, the loop must not use `short *` or `int *` pointers. See **C/C++ Check-Sum Function (Item 2, Item 9)** on page 504.

10. Storing the length of a byte string in its first two bytes

If the string can begin at an arbitrary address, the two bytes in which the length is stored must be stored and accessed as separate bytes, not as a single short integer. See:

- **C/C++ Invalid Cast From char to Integer Pointer (Item 2, Item 10)** on page 503
- **C/C++ Invalid Cast From char to Integer Pointer (Item 2, Item 10)** on page 503
- **C/C++ Pointer Union (Item 3, Item 10)** on page 503

11. Ending a byte string with multiple zero-filled bytes

The zero-filled bytes must be inserted by separate single-byte storage operations, not by a single `short` storage operation.

C/C++ Null Pointer Check (Item 1b)

Change this:

```
struct listnode {
int *kind;
...
struct listnode *next;
};
struct listnode *listhead, *node;

node = listhead;
while (*node->kind == 4 /* used too soon */ && node != NULL) {
...
node = node->next;
}
```

To this:

```
struct listnode {
int *kind;
...
struct listnode *next;
};
struct listnode *listhead, *node;

node = listhead;
while (node != NULL && *node->kind == 4) {
...
node = node->next;
}
```

C/C++ Invalid Cast From char to Integer Pointer (Item 2, Item 10)

Change this:

```
/* extract 16-bit length from front of long string: */
    unsigned char *name;
    lth = * (unsigned short *) name;  /* misalignment traps here */
    name += 2;
```

To this:

```
/* extract 16-bit length from front of long string: */
    unsigned char *name;
    lth = (name[0] << 8) | name[1]
    name += 2;
```

C/C++ Invalid Cast From char to Integer Pointer (Item 2, Item 10)

Change this:

```
/* insert 16-bit length at front of long string: */
    unsigned char *name;
    unsigned short lth;
    * (unsigned short *) name = lth;
```

To this:

```
/* insert 16-bit length at front of long string: */
    unsigned char *name;
    unsigned short lth;
    name[0] = lth >> 8;
    name[1] = lth;
```

C/C++ Pointer Union (Item 3, Item 10)

Change this:

```
/* type cast done via union: */
    union { unsigned char  *bytes;
            unsigned short *shorts; } u;
    u.bytes = name;
    lth = * u.shorts; /* misalignment traps here */
```

To this:

```
lth = (name[0] << 8) | name[1];
```

C/C++ Structure With Implicit Filler Bytes (Item 5)

```
struct elem {
char  a;    /* offset 0 */
/* implicit filler byte added before b */
short b;    /* offset 2 */
char  c;    /* offset 4 */
/* implicit trailing filler byte added after c */
/* with filler bytes, total size is 6 bytes */
};
struct elem table[10];  /* each table element takes 6 bytes, not 4 */
```

C/C++ External Structure With Misaligned Parts (Item 8)

Change this:

```
/* An interface as declared and used on 8-bit CPUs: */
    struct {
        short a;   /* offset 0 */
        char  b;   /* offset 2 */
        short c;   /* offset 3 */  /* moves to offset 4 on TNS */
    } s;
    i = s.c;
    s.c = j;
```

To this:

```
    struct {
        short a;   /* offset 0 */
        char  b;   /* offset 2 */
        unsigned char c[2];  /* represents an unaligned short */
    } s;

#define get_short(var)      ((short) ((var[0] << 8) | var[1]))
#define put_short(var,val)  {var[0] = (val) >> 8;  var[1] = (val);}
    get_short(s.c);
    put_short(s.c, j);
```

C/C++ Check-Sum Function (Item 2, Item 9)

Change this:

```
char *name;
int lth;
unsigned short checksum = 0;

while (lth >= 2) {
    checksum += * (short *) name;   /* misalignment traps here */
    name += 2;   lth -= 2;
}
if (lth > 0) checksum += *name << 8;
```

To this:

```
char *name;
int lth;
unsigned short checksum = 0;

while (lth >= 2) {
    checksum += (name[0] << 8) | name[1];
    name += 2;   lth -= 2;
}
if (lth > 0) checksum += *name << 8;
```

LP64 Data Model

Support for the LP64 Data Model

In H06.24 and later H-series RVUs and in J06.13 and later J-series RVUs, and L-series, OSS supports a new LP64 data model, along with the existing default ILP32 data model. The LP64 data model is used to create 64-bit OSS user processes. The C/C++ compiler allows the user to compile an OSS application in ILP32 as well as in LP64 mode.

In the ILP32 data model, the data types: `int`, `long`, and `pointers` are 32-bit in size. Whereas, in the LP64 data model, the data types: `long` and `pointers` are 64-bit and `ints` continue to be 32-bit in size. **Size (in bits) for the various C/C++ Data Types** summarizes the size (in bits) of the various C/C++ data types.

Table 67: Size (in bits) for the various C/C++ Data Types

Data Types	ILP32 data model	LP64 data model
char	8	8
short	16	16
int	32	32
long	32	64
long long	64	64
void *	32	64
enum	32	32
& (address of)[1]	32	64
float	32	32
double	64	64

[1] The address taken from an item referenced via a 64-bit pointer produces a 64-bit address.

Selecting the Data Model on the Command-Line

The default data model is ILP32. You can override the default with the following command-line directives

`-Wlp64`

is the `OSS` and `Windows` command-line directive to specify the LP64 data model.

`LP64`

is the Guardian command-line directive to specify the LP64 data model.

`-Wilp32`

is the `OSS` and `Windows` command-line directive to specify the ILP32 data model, this is the default option.

`ILP32`

is the Guardian command-line directive to specify the ILP32 data model, this is the default option.

Command Line Restrictions

The LP64 data model does not support the following CCOMP options :

`SQL`

`SYSTYPE GUARDIAN`

The LP64 data model does not support the following CPPCOMP options :

`VERSION2`

`SYSTYPE GUARDIAN`

`–Wlp64` does not support the following c89 options :

`-Wsql`

`-Wsystype=guardian`

`-Wversion2`

`–Wlp64` does not support the following c99 options :

`-Wsystype=guardian`

Pointer Modifiers

The pointer modifiers specify the size of a pointer. The `_ptr64` modifier is used to declare a 64-bit pointer and the `_ptr32` modifier is used to declare a 32-bit pointer. These modifiers may be used in both ILP32 and LP64 modes.

The grammar for the new pointer modifiers are:

```
pointer:
   [ modifier-list ] * [ type-qualifier-list]
   [ modifier-list ] * [ type-qualifier-list] pointer

modifier-list:
   modifier [ modifier-list ]

modifier:
   _far | _near | _baddr | _waddr | _procaddr | _ptr32 | _ptr64
```

At most, one modifier is specified for a pointer type.

C++ does not allow function overloading based entirely on differences in return types. In C++ mode, pointer modifiers alone must not be used to overload functions.

The compiler returns an error on attempts to overload functions based on only pointer size differences.

Example

```
char _ptr64 *Ptr1;   /* Ptr1 is a 64-bit pointer to a character. /
char _ptr32 *Ptr2;   /* Ptr2 is a 32-bit pointer to a character. /
char        *Ptr3;   /* Size of Ptr3 is the default size for the
                        compilation’s data model. */
typedef void _far _ptr64 * Ptr64;   /* Error, cannot specify multiple
```

```
                                       modifiers. */
void _ptr32 *foo (void);
void _ptr64 *foo(void) {}    /* Error, mismatched modifiers on return type. */
void Foo (void *ptr);
void Foo (void _ptr64 *ptr);  /* Error, mismatched modifiers on parameter
                                 type. */
```

Accessing 32-bit memory in the LP64 Data Model

The following functions are used in a LP64 data model process to access a secondary 32-bit addressable heap space:

`calloc32()`

`free32()`

`malloc32()`

`realloc32()`

`heap_check32()`

`heap_check_always32()`

For more details, see *Open System Services Library Calls Reference Manual*.

Data Model Pragmas

`#pragma p32_shared` and `#pragma p64_shared`

These pragmas apply to `class`, `struct`, or `union` definitions. For C, they affect pointers declared as fields. For C++, they apply to pointers declared (implicitly or explicitly) as members, to function parameters (including the implicit “this” parameter) and to any pointers used in the member function definitions.

It does not allow the components whose layout or size are not data model invariant. It means that, `long` is disallowed as a component and any `class`, `struct` or `union` component must be declared as either `p32_shared` or `p64_shared`.

These pragmas immediately precede the `class`, `struct`, or `union` definition. They are not applied globally and to any textually nested `class`, `struct`, or `union` definitions. They are not used as an argument to pragmas `push` or `pop`.

`p32_shared`

It indicates that within the definition, the unqualified pointer declarations are 32-bit pointers – regardless of the data model. Any base class of a p32_shared class must also be a p32_shared class.

`p64_shared`

It indicates that within the definition, the unqualified pointer declarations are 64-bit pointers – regardless of the data model. Any base class of a p64_shared class must also be a p64_shared class.

`#pragma p32_shared` Example

```
struct s0{
  int        *Ptr1;   /* Size depends on data model. */
  int _ptr32 *Ptr2;   /* This is a 32-bit pointer. */
  int _ptr64 *Ptr3;   /* This is a 64-bit pointer. */
};
#pragma p32_shared
struct s1{
  int        *Ptr1;   /* This is a 32-bit pointer. */
  int _ptr32 *Ptr2;   /* This is a 32-bit pointer. */
  int _ptr64 *Ptr3;   /* This is a 64-bit pointer. */
};
#pragma p32_shared
struct s2{
  long   field1;   /* Error, long not allowed as a component. */
  long  *field2;   /* Ok, this is a 32-bit pointer to a long. /
  struct s0 S0;    /* Error,not p32_shared or p64_shared. /
  struct s1 S1;    /* Ok. */
};
```

#pragma p64_shared Example

```
#pragma p64_shared
struct s1{
  int        *Ptr1;   /* This is a 64-bit pointer. */
  int _ptr32 *Ptr2;   /* This is a 32-bit pointer. */
  int _ptr64 *Ptr3;   /* This is a 64-bit pointer. */
};
#pragma p64_shared
struct s2{
  long   field1;   /* Error, long not allowed as a component. */
  long  *field2;   /* Ok, this is a 64-bit pointer to a long. */
};
```

Data Model Macros

Data Model Macros lists predefined macros that are set based on the data model specified on the compiler command-line. They are used when writing source code that is intended to be compiled in both data models.

Table 68: Data Model Macros

Macro	What it defines
__ILP32	Defined automatically, when the data model specified (explicitly or implicitly) is ILP32. Its value is 1.
__LP64	Defined automatically, when the data model specified is LP64. Its value is 1.

New Intrinsics <builtin.h>

New Intrinsics defined in header <builtin.h> are as follows:

is_32bit_addr

```
int _is_32bit_addr(void _ptr64 *);
```

If the passed pointer is a sign-extended 32-bit address, the function returns a non-zero value.

_ptr64_to_ptr32

The following new builtin insures that a 64-bit pointer safely truncates to a 32-bit pointer:

```
void _ptr32 * _ptr64_to_ptr32(void _ptr64 *);
```

If the passed pointer is a sign-extended 32-bit pointer, its low-order 32 bits are returned. Otherwise, a run-time trap occurs.

NOTE:

This intrinsic is not influenced by `#pragma no_overflowtraps.`

Migration Warnings

The compiler has the ability to detect a number of 64-bit porting issues.

For some issues, the compiler returns a diagnostic by default. For example, any direct assignment from a 64-bit pointer to a 32-bit pointer returns:

```
error(611): a value of type "<longpointertype>" cannot be assigned to
  an entity of type "<shortpointertype>"
```

In addition, the compiler provides a feature to return additional warnings. These warnings detect valid C/C ++ code that potentially behaves in an unexpected fashion when code designed for the ILP32 data model is compiled using the LP64 data model.

This feature is controlled by a command line directive. On Windows and OSS, the option is --`Wmigration_check=32to64`. On Guardian, the option is `migration_check 32to64`.

An example of one of these diagnostics is:

Any conversion (using either a cast or a direct assignment) from a `long` to an `int` returns:

```
warning(2412): 64 bit migration: type conversion may truncate value
```

Another example is:

Any conversion either from a pointer to `long` to a pointer to `int`, or from a pointer to `int` to a pointer to `long`, returns:

```
warning(2414): 64 bit migration: type conversion may cause target of pointers
to have a different size
```

HPE recommends that you compile the code with the migration check warnings enabled, and each warning examined, before compiling it with `–Wlp64` option.

Changed Features

pragma shared2/shared8

When pragma `shared2/shared8` is compiled with the LP64 data model, the compiler issues an error for components whose type is `long` or whose type is a pointer (without an explicit size modifier). These components are grandfathered for the ILP32 mode.

Websites

General websites

Hewlett Packard Enterprise Information Library

www.hpe.com/info/EIL

Hewlett Packard Enterprise Support Center

www.hpe.com/support/hpesc

Contact Hewlett Packard Enterprise Worldwide

www.hpe.com/assistance

Subscription Service/Support Alerts

www.hpe.com/support/e-updates

Software Depot

www.hpe.com/support/softwaredepot

Customer Self Repair

www.hpe.com/support/selfrepair

Manuals for L-series

http://www.hpe.com/info/nonstop-ldocs

Manuals for J-series

http://www.hpe.com/info/nonstop-jdocs

For additional websites, see **Support and other resources**.

Support and other resources

Accessing Hewlett Packard Enterprise Support

- For live assistance, go to the Contact Hewlett Packard Enterprise Worldwide website:

  http://www.hpe.com/assistance

- To access documentation and support services, go to the Hewlett Packard Enterprise Support Center website:

  http://www.hpe.com/support/hpesc

Information to collect

- Technical support registration number (if applicable)
- Product name, model or version, and serial number
- Operating system name and version
- Firmware version
- Error messages
- Product-specific reports and logs
- Add-on products or components
- Third-party products or components

Accessing updates

- Some software products provide a mechanism for accessing software updates through the product interface. Review your product documentation to identify the recommended software update method.
- To download product updates:

  Hewlett Packard Enterprise Support Center

  www.hpe.com/support/hpesc

  Hewlett Packard Enterprise Support Center: Software downloads

  www.hpe.com/support/downloads

  Software Depot

  www.hpe.com/support/softwaredepot

- To subscribe to eNewsletters and alerts:

  www.hpe.com/support/e-updates

- To view and update your entitlements, and to link your contracts and warranties with your profile, go to the Hewlett Packard Enterprise Support Center **More Information on Access to Support Materials** page:

www.hpe.com/support/AccessToSupportMaterials

ⓘ **IMPORTANT:** Access to some updates might require product entitlement when accessed through the Hewlett Packard Enterprise Support Center. You must have an HPE Passport set up with relevant entitlements.

Customer self repair

Hewlett Packard Enterprise customer self repair (CSR) programs allow you to repair your product. If a CSR part needs to be replaced, it will be shipped directly to you so that you can install it at your convenience. Some parts do not qualify for CSR. Your Hewlett Packard Enterprise authorized service provider will determine whether a repair can be accomplished by CSR.

For more information about CSR, contact your local service provider or go to the CSR website:

http://www.hpe.com/support/selfrepair

Remote support

Remote support is available with supported devices as part of your warranty or contractual support agreement. It provides intelligent event diagnosis, and automatic, secure submission of hardware event notifications to Hewlett Packard Enterprise, which will initiate a fast and accurate resolution based on your product's service level. Hewlett Packard Enterprise strongly recommends that you register your device for remote support.

If your product includes additional remote support details, use search to locate that information.

Remote support and Proactive Care information

HPE Get Connected

www.hpe.com/services/getconnected

HPE Proactive Care services

www.hpe.com/services/proactivecare

HPE Proactive Care service: Supported products list

www.hpe.com/services/proactivecaresupportedproducts

HPE Proactive Care advanced service: Supported products list

www.hpe.com/services/proactivecareadvancedsupportedproducts

Proactive Care customer information

Proactive Care central

www.hpe.com/services/proactivecarecentral

Proactive Care service activation

www.hpe.com/services/proactivecarecentralgetstarted

Warranty information

To view the warranty for your product, see the *Safety and Compliance Information for Server, Storage, Power, Networking, and Rack Products* document, available at the Hewlett Packard Enterprise Support Center:

www.hpe.com/support/Safety-Compliance-EnterpriseProducts

Additional warranty information

HPE ProLiant and x86 Servers and Options

www.hpe.com/support/ProLiantServers-Warranties

HPE Enterprise Servers

www.hpe.com/support/EnterpriseServers-Warranties

HPE Storage Products

www.hpe.com/support/Storage-Warranties

HPE Networking Products

www.hpe.com/support/Networking-Warranties

Regulatory information

To view the regulatory information for your product, view the *Safety and Compliance Information for Server, Storage, Power, Networking, and Rack Products*, available at the Hewlett Packard Enterprise Support Center:

www.hpe.com/support/Safety-Compliance-EnterpriseProducts

Additional regulatory information

Hewlett Packard Enterprise is committed to providing our customers with information about the chemical substances in our products as needed to comply with legal requirements such as REACH (Regulation EC No 1907/2006 of the European Parliament and the Council). A chemical information report for this product can be found at:

www.hpe.com/info/reach

For Hewlett Packard Enterprise product environmental and safety information and compliance data, including RoHS and REACH, see:

www.hpe.com/info/ecodata

For Hewlett Packard Enterprise environmental information, including company programs, product recycling, and energy efficiency, see:

www.hpe.com/info/environment

Documentation feedback

Hewlett Packard Enterprise is committed to providing documentation that meets your needs. To help us improve the documentation, send any errors, suggestions, or comments to Documentation Feedback (**docsfeedback@hpe.com**). When submitting your feedback, include the document title, part number, edition, and publication date located on the front cover of the document. For online help content, include the product name, product version, help edition, and publication date located on the legal notices page.

HPE C Implementation-Defined Behavior

Implementation-Defined Behavior of Native C

The ISO standard for C allows implementations to vary in specific instances. This subsection describes the implementation-defined behavior of native C. This subsection corresponds to Annex G.3 of the ISO C standard or Appendix F of the ANSI C standard.

G.3.1 Translation

The form of the diagnostic messages displayed by the native compilers is such that the source line is first displayed, followed by a line that indicates the location, and, finally, a line of the form: file name, line: diagnostic-type: diagnostic message. For example:

```
FILE *fp;)
^
“/usr/people/bj/test/c1.c”, line 2, error(114): identifier “FILE” is undefined
```

G.3.2 Environment

The arguments to main() are treated:

Argument	Description
`argv [0]`	the name of the executable file
`argv [1] ... argv [argc-1]`	command line parameters
`argv [argc]`	a null pointer

An interactive device is a video display terminal.

G.3.3 Identifiers

An identifier without external linkage has a maximum of 230 significant initial characters.

An identifier with external linkage has a total of 127 significant characters.

Case distinctions are significant in an identifier with external linkage.

G.3.4 Characters

The shift states used for the encoding of multibyte characters are:

- The number of bits in a character in the execution character set is 8, but only the lower 7 are significant.
- The mapping of characters of the source character set to members of the execution character set is one to one.
- There are no invalid characters or escape sequences in the basic execution character set.
- The value of an integer character constant that contains more than one character or a wide character that contains more than one multibyte character, for character constants with 2 to 4 characters, is:

2 characters "ab"
'a' * 256 + (unsigned) 'b'

3 characters "abc"
'a' * 65536 + (unsigned) 'b' * 256 + (unsigned) 'c'

4 characters "abcd"
'a' * 16777216 + (unsigned) 'b' * 65536+ (unsigned) 'c' * 256 + (unsigned) 'd'

- The C locale is used to convert multibyte characters into corresponding wide character codes. The value of the wide character is equal to the value of the first byte in the multibyte sequence (whose value is taken as an unsigned value).
- A plain `char` has the same range of values as `unsigned char`.

G.3.5 Integers

Converting an integer to a shorter signed integer causes a representation change by discarding the high order bits. Converting an unsigned integer to a signed integer of equal length does not cause a representation change; however, the converted value may be negative. The `unsigned long long` data type is supported for TNS/R, TNS/E, and TNS/X-targeted compilations.

Bitwise operations on signed integers produce signed results, represented in two's complement. How the value is interpreted depends on whether the sign bit is on or off after the operation. The operation is performed on the data as though the values were unsigned.

When the operator is % (remainder on integer division) these are true:

- If the dividend is negative and the divisor is positive, the result is negative.
- If the dividend is positive and the divisor is negative, the result is negative.
- If both are negative the result is negative.

A right shift of a negative signed integral type causes the sign bit to be replicated. (The shift is an arithmetic shift instead of a logical shift.)

G.3.6 Floating Point

Native C and C++ support two floating-point formats (IEEE and Tandem), beginning at the G06.06 release. Tandem floating-point is further described in **G.3.6 Floating Point** on page 532.

The conversion behavior for IEEE floating-point types is:

Table 69: Conversion Behavior for IEEE Floating-Point Format

Conversion	IEEE Floating-Point
From fixed-point to floating-point	Rounds to the nearest value according to the current IEEE floating-point rounding mode
From floating-point to fixed-point	Truncates to the nearest representable value
From a floating-point number to a narrower floating-point number	Rounds to the nearest value according to the current IEEE floating-point rounding mode

The default IEEE rounding mode is to round to the nearest representable value. Where two values are equally close, the even value (the one with a zero in the least significant fraction bit) is chosen. For more details about the FP_IEEE_ROUND_GET_ and FP_IEEE_ROUND_SET_ routines, see the *Guardian Procedure Calls Reference Manual*.

Tandem floating-point format and IEEE floating-point format have different ranges and precision. Therefore, a value that might convert exactly if one format is used might not convert exactly if the other format is used. For more details about ranges and precision of each format, see **IEEE Floating-Point Arithmetic** on page 82 and pragma **IEEE_FLOAT** on page 248.

G.3.7 Arrays and Pointers

`size_t` is defined in `stddef.h` to be `unsigned long`.

Converting a pointer to a signed integer of equal length does not cause a representation change. Converting an integer to a shorter signed integer causes a representation change by discarding the high order bits.

`ptrdiff_t` is defined in `stddef.h` to be `long`.

G.3.8 Registers

The `register` storage class specifier cannot be used with structure or array declarations. A register variable may be changed to a nonregister type or a nonregister type changed to register by the optimizer.

G.3.9 Structures, Unions, Enumerations, and Bit Fields

A union is considered as a block of memory the size of the union. If a member of a union has a value stored in it and is subsequently accessed using a member of a different type the result is defined as the value of the accessed type at that block of memory. If the size of the type stored is smaller than the accessed type and the accessed value is beyond the stored type, the result is undefined. If the type stored is a structure with holes, and the accessed value overlaps any of the holes, the value is undefined.

Each member of a structure is aligned on the boundary required by its type. Padding is added between members as necessary.

A plain `int` bit field is a `signed int` bit field.

Bits within an integer bit field are allocated most significant bit first.

A bit field cannot straddle a word boundary. If the bit field is a member of a structure with alignment `SHARED2`, and the size of the bit field is no greater than 16 bits, it is further restricted not to straddle a half word boundary.

The values of an enumeration declaration are type `int`.

G.3.10 Qualifiers

Each time a value is needed from a volatile object, a read access is made to it. Each time the value needs to be written, a write access is made. This rule ensures that volatile objects are accessed in the same way as in the abstract semantics. However, there is one exception: when a volatile bit field is written to, hardware constraints may force a read to occur before the write, to read the parts of the storage unit that are not changed in the write. Volatile bit fields should be used with caution.

G.3.11 Declarators

An arithmetic, structure, or union type can have 12 declarators modifying it. The maximum number of declarators allowed is limited only by the amount of available memory at compilation time.

G.3.12 Statements

The maximum number of case values allowed in a switch statement is limited only by the amount of available memory at compilation time.

G.3.13 Preprocessing Directives

The value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. A single-character character constant is an unsigned character and therefore cannot be negative.

When the include file is specified as "*filename*" :

- If the compiler is running in the Guardian environment, the current subvolume is searched first
- If the compiler is running in the OSS environment, the current directory is searched first

When an include file is specified as <*filename*>:

- If the compiler is running in the Guardian environment, the only subvolume searched is the compiler's subvolume (usually, `$system.system`)
- If the compiler is running in the OSS environment, the only directory searched is `/usr/include`.

The SSV pragma can be used to modify the subvolume searched when running under Guardian, and the -I option can be used to modify the search directories when running under OSS.

For the descriptions of the pragmas supported by native C, see **Compiler Pragmas** on page 193.

When the date or time of translation is not available, the definitions of the __DATE__ and __TIME__ macros are January 1, 1970 and 00:00:00, respectively.

G.3.14 Library Functions

OL is the null pointer constant to which the macro NULL expands.

For more details on how to recognize the diagnostic that is printed by the assert() function and information on the termination behavior of the assert() function, see the assert() function in the *Guardian Native C Library Calls Reference Manual.*

These characters are tested for by the `isalnum()`, `isalpha()`, `iscntrl()`, `islower()`, `isprint()`, and `isupper()` functions:

- `isalnum()` returns true for characters a–z, A–Z, 0–9
- `isalpha()` returns true for characters a–z, A–Z
- `iscntrl()` returns true for values 0–31, 127
- `islower()` returns true for characters a–z
- `isprint()` returns true for values 32–126
- `isupper()` returns true for characters A–Z

An undefined value is returned by the mathematics functions after a domain error, and `errno` is set to `EDOM`.

The mathematics functions set the integer expression `errno` to the value of the macro `ERANGE` on underflow range errors.

When the fmod() function has a second argument of zero, `fmod()` returns the value of the first argument (that is zero is treated as one).

Signals

For the set of signals for the signal() function and a description of the parameters and the usage of each signal, see the `signal(4)` reference page online or in the *Open System Services System Calls Reference Manual*.

For each signal recognized by the `signal()` function, at program startup the handler `SIG_DFL` is registered for the all the signals by the C runtime.

The default handling is reset if a `SIGILL` signal is received by a handler specified to the `signal()` function.

Streams and Files

The last line of a text stream does not require a terminating newline character. The file is written with the characters requested.

The space characters that are written out to a text stream immediately before a newline character appear when the stream is read back in.

Zero null characters may be appended to data written to a binary stream.

The file position indicator of an append mode stream is initially positioned at the start of the file.

A write on a text stream does not cause the associated file to be truncated beyond that point.

Full buffering best describes the file buffering of the native C run-time.

A zero-length file can actually exist.

These are the rules for composing a valid file name:

- In the Guardian environment, the file name is formed as specified by the Guardian file-name format, which is composed of system name, volume name, subvolume name and file name, each separated by a period(.). Except for the file name, all the other fields can be omitted, then the default subvolume is the present working subvolume.
- In the OSS environment, the newline file name must be a valid OSS filename.

The same file can be opened simultaneously multiple times, as long as it is not for writing.

The effect of the `remove()` function on an open file is that it returns a nonzero value and does not remove the file.

If a file with a new name already exists prior to a call to the `rename()` function, it returns a nonzero value without renaming the file.

The output for `%p` conversion in the `fprintf()` function formats the value of a pointer to a void argument using the format specified by the `x` conversion code.

The input for `%p` conversion in the `fscanf()` function matches an unsigned hexadecimal integer that uses the lowercase letters `a` through `f` to represent the digits 10 through 15. The corresponding *obj_ptr* must be a pointer to void. Consequently, the integer is interpreted as a pointer to void.

The hyphen character (`-`) is treated as just another character in the scan list if it is not the first or the last character in the scan list.

tmpfile()

An open temporary file is removed if the program terminates abnormally.

errno

The macro `errno` is set to 4009 by the `fgetpos()` or `ftell()` function on failure.

A call to the `perror()` function prints the textual error message corresponding to the value of the errno, optionally preceded by a specified string.

Memory

The behavior of the `calloc()`, `malloc()`, or `realloc()` function if the size requested is zero is:

`calloc()`, `malloc()`, and `realloc()` return a NULL pointer.

abort()

When the `abort()` function is called, all open and temporary files are closed. The temporary files are deleted.

exit()

The status returned by the `exit()` function if the value of the argument is other than zero, EXIT_SUCCESS, or EXIT_FAILURE is:

The `exit()` function does not return.

getenv()

The environment names are always in uppercase. These four run-time parameters are always present:

Parameter	Description
`STDIN`	Gives the name of the standard input file, `stdin`.
`STDOUT`	Gives the name of the standard output file, `stdout`.
`STDERR`	Gives the name of the standard error file, `stderr`.
`DEFAULTS`	Gives the default volume and subvolume names used to qualify partial file names.

To alter the Guardian environment list obtained by a call to the getenv() function, use the PARAM command, the syntax for which is:

`PARAM` *`param_name param_setting`*

param_name

is the run-time environment parameter name

param_setting

is the string that will be returned when getenv is called with *param_name* as its parameter.

system()

The string passed to the `system()` function is either a NULL string or an ASCII string that can be processed by the command processor.

strerror()

The format of the error message returned by the `strerror()` function is an ASCII string.

errno Values and Corresponding Messages lists the contents of the error message strings returned by a call to the strerror() function:

Table 70: errno Values and Corresponding Messages

errno	Messages
EPERM	Not owner, permission denied
ENOENT	No such file or directory
ESRCH	No such process or table entry
EINTR	Interrupted system call
EIO	I/O error
ENXIO	No such device or address
E2BIG	Argument list too long
ENOEXEC	Exec format error
EBADF	Bad file descriptor
ECHILD	No children
EAGAIN	No more processes
ENOMEM	Insufficient user memory
EACCES	Permission denied
EFAULT	Bad address
EBUSY	Mount device busy
EEXIST	File already exists
EXDEV	Cross-device link
ENODEV	No such device
ENOTDIR	Not a directory
EISDIR	Is a directory
EINVAL	Invalid function argument
ENFILE	File table overflow
EMFILE	Maximum number of files already open

Table Continued

errno	Messages
ENOTTY	Not a typewriter
EFBIG	File too large
ENOSPC	No space left on device
ESPIPE	Illegal seek
EROFS	Read only file system
EMLINK	Too many links
EPIPE	Broken pipe or no reader on socket
EDOM	Argument out of function's domain
ERANGE	Value out of range
EDEADLK	Deadlock condition
ENOLCK	No record locks available
ENODATA	No data sent or received
ENOSYS	Function not implemented
EWOULDBLOCK	Operation would block
EINPROGRESS	Operation now in progress
EALREADY	Operation already in progress
ENOTSOCK	Socket operation on non-socket
EDESTADDRREQ	Destination address required
EMSGSIZE	Message too long
EPROTOTYPE	Protocol wrong type for socket
ENOPROTOOPT	Protocol not available
EPROTONOSUPPORT	Protocol not supported
ESOCKTNOSUPPORT	Socket type not supported
EOPNOTSUPP	Operation not supported on socket
EPFNOSUPPORT	Protocol family not supported

Table Continued

errno	Messages
EAFNOSUPPORT	Address family not supported
EADDRINUSE	Address already in use
EADDRNOTAVAIL	Can't assign requested address
ENETDOWN	Network is down
ENETUNREACH	Network is unreachable
ENETRESET	Network dropped connection on reset
ECONNABORTED	Software caused connection abort
ECONNRESET	Connection reset by remote host
ENOBUFS	No buffer space available
EISCONN	Socket is already connected
ENOTCONN	Socket is not connected
ESHUTDOWN	Can't send after socket shutdown
ETIMEDOUT	Connection timed out
ECONNREFUSED	Connection refused
EHOSTDOWN	Host is down
EHOSTUNREACH	No route to host
ENAMETOOLONG	File name too long
ENOTEMPTY	Directory not empty
EHAVEOOB	Out-of-band data available
EBADSYS	Invalid socket call
EBADFILE	File type not supported
EBADCF	Not a C file
ENOIMEM	Insufficient internal memory
EBADDATA	Invalid data in buffer
ENOREPLY	No reply in buffer

Table Continued

errno	Messages
EPARTIAL	Partial buffer received
ESPIERR	Interface error from SP I
EVERSION	Version mismatch
EXDRDECODE	XDR encoding error
EXDRENCODE	XDR decoding error

G.4 Locale Behavior

The local time zone and daylight-saving time depend on the system location.

The era for the clock function "Locale-specific Behavior" is January 1, 1970 GMT.

No characters have been added to the execution set required by the ISO/ANSI C standard.

The direction of printing is left to right, and top to bottom.

The decimal point character is a period (`.`).

G.5 Common Extensions

There are no common extensions to the formats for time and date.

Multibyte Characters and Wide Characters

Multibyte characters and wide characters support Asian alphabets that often contain a very large number of characters. The Guardian native C run-time library functions, except for the `strcoll()` and `strxfrm()` functions, support these character sets: Tandem Kanji, Chinese Big 5, Chinese PC, Hangul and KSC5601.

Discussion of multibyte characters applies only to the Guardian environment. For more details on multibyte characters in the Open System Services (OSS) environment, see the *Software Internationalization Manual*.

The Guardian native C run-time library functions `mblen()`, `mbtoc()`, `mbtowcs()`, `wctomb()`, and `wctombs()` do not support multibyte characters for programs that use the 32-bit (or wide) data model as described in this section. Guardian programs that use the 32-bit data model must use the Guardian system procedures that support multibyte characters instead. For more details, see the *Guardian Programmer’s Guide*.

The default character set supported by a system is configured at system installation time and cannot be changed during program execution. The Guardian procedure MBCS_DEFAULTCHARSET_ returns the identifier of the default character set. The *Guardian Procedure Calls Reference Manual* describes this system procedure in detail.

The internal representation of the characters of these languages is HPE internal and might not conform to any ISO standard. HPE can choose to change this internal representation at any time.

Multibyte Characters

- The basic difficulty in an Asian environment is the huge number of ideograms that are needed for I/0, for example Chinese characters. To work within the constraints of usual computer architectures, these ideograms are encoded as sequences of bytes. The associated operating systems, application programs, and terminals understand these byte sequences as individual ideograms. Moreover, all these encodings allow intermixing of regular single-byte C characters with the ideogram byte sequences.

- The term “multibyte character” denotes a byte sequence that encodes an ideogram. The byte sequence contains one or more codes where each code can be represented in a C character data type: char, signed char, or unsigned char. All multibyte characters are members of the so-called extended character set. A regular single-byte C character is just a special case of a multibyte sequence where the sequence has a length of one.

Wide Characters

Some of the inconvenience of handling multibyte characters is eliminated if all characters are of a uniform number of bytes or bits. A 16-bit integer value is used to represent all members because there can be thousands or tens of thousands of ideograms in an Asian character set.

Wide characters are integers of type `wchar_t`, defined in the headers `stddef.h` and `stdlib.h` as:

```
typedef unsigned short wchar_t;
```

Such an integer can represent distinct codes for each of the characters in the extended character set. The codes for the basic C character set have the same values as their single-character forms.

Relationship Between Multibyte and Wide Characters

- Multibyte characters are convenient for communicating between the program and the outside world.
- Wide characters are convenient for manipulating text within a program.
- The fixed size of wide characters simplifies handling both individual characters and arrays of characters.

`MB_CUR_MAX` Macro

The `MB_CUR_MAX` macro specifies the maximum number of bytes used in representing a multibyte character in the current locale (category `LC_CTYPE`). The `MB_CUR_MAX` macro is defined in the header STDLIBH as:

```
#define MB_CUR_MAX  2
```

Conversion Functions

The run-time library functions that manage multibyte characters and wide characters are:

Function	Description
`mblen()`	Determines the length of a multibyte character.
`mbtowc()`	Converts a multibyte character to a wide character.
`wctomb()`	Converts a wide character to a multibyte character.

Table Continued

Function	Description
`mbstowcs()`	Converts a string of multibyte characters to a string of wide characters.
`wcstombs()`	Converts a string of wide characters to a string of multibyte characters.

Alignment Issues

Native C and C++ considers objects of integral types to exist only on word boundaries. Consequently, it is invalid to use an odd-byte address to access such an object. On TNS/R, TNS/E, or TNS/X systems, the results of using an integer type extended pointer containing an odd-byte address are undefined. The code might continue executing or trap. Therefore, it is important for you to ensure that all extended pointers contain addresses that are even except for pointers to `char`. Extended pointers are those of type long `int` or those of type `int` with the 32-bit (wide) data model in effect, in which case an int is represented by 32 bits.

Translation Limits for Native C Compilers

Native C uses dynamic data structures, so some program components are limited by the amount of available memory. This list indicates minimums that are guaranteed (that is, a program that meets but does not exceed each minimum is guaranteed to compile). However, if a program significantly exceeds one or more minimums, it is possible to run out of memory and to receive an error message on a component whose minimum has not been reached. The limits differ for the TNS/R, TNS/E, and TNS/X native C compilers.

TNS/R Native C Compiler

- Compound statements, iteration control statements, and selection control statements can be nested 15 levels.
- Conditional `#include` directives can be nested 8 levels.
- Arithmetic, structure, union, or incomplete type declarations can have 12 pointer, array, and function declarators modifying them.
- A declaration can have 31 nested levels of parenthesized declarators.
- An expression can have 32 nested levels of parenthesized expressions.
- An internal identifier or macro name can have 31 significant characters.
- An external identifier can have 31 significant initial characters.
- A single translation unit can have 511 external identifiers.
- A block can have 127 identifiers declared with block scope.
- A single translation unit can have 1024 macro identifiers defined simultaneously.
- A macro definition can have 31 parameters and a macro invocation 31 arguments.
- A logical source line can have 509 characters.
- A string literal or wide string literal can have 509 characters (after string concatenation).
- An object can consist of 32767 bytes.

- A `switch` statement can have 257 case labels (excluding any nested switch statements).
- A single `struct` or `union` can have 127 members.
- A single enumeration can have 127 enumeration constants.
- A single structure declaration can have 15 levels of nested structure or union definitions.

TNS/E Native C Compiler

- Compound statements, iteration control statements, and selection control statements can be nested to at least 127 levels.
- Conditional `#include` directives can be nested 63 levels.
- Arithmetic, structure, union, or incomplete type declarations can have at least 12 pointer, array, and function declarators modifying them.
- A declaration can have at least 63 nested levels of parenthesized declarators.
- An expression can have at least 63 nested levels of parenthesized expressions.
- An internal identifier or macro name can have 63 significant characters.
- An external identifier can have 31 significant initial characters.
- A single translation unit can have at least 4095 external identifiers.
- A block can have at least 511 identifiers declared with block scope.
- A single translation unit can have at least 127 macro identifiers defined simultaneously.
- A macro definition can have at least 127 parameters and a macro invocation at least 31 arguments.
- A logical source line can have at least 4095 characters.
- A string literal or wide string literal can have at least 4095 characters (after string concatenation).
- An object can consist of at least 32767 bytes.
- A constant or variable can consist of at least 65535 bytes.
- #included files can have 15 nested levels.
- A switch statement can have 1023 case labels (exclusing any nested switch statements).
- A single struct or union can have at least 1023 members.
- A `switch` statement can have at least 1023 case labels (excluding any nested switch statements).
- A single `struct` or `union` can have at least 1023 members.
- A single enumeration can have at least 1024 enumeration constants.
- A single structure declaration can have at least 63 levels of nested structure or union definitions.

TNS/X Native C Compiler

- Compound statements, iteration control statements, and selection control statements can be nested to at least 127 levels.
- Conditional `#include` directives can be nested 63 levels.

- Arithmetic, structure, union, or incomplete type declarations can have at least 12 pointer, array, and function declarators modifying them.
- A declaration can have at least 63 nested levels of parenthesized declarators.
- An expression can have at least 63 nested levels of parenthesized expressions.
- An internal identifier or macro name can have 63 significant characters.
- An external identifier can have 31 significant initial characters.
- A single translation unit can have at least 4095 external identifiers.
- A block can have at least 511 identifiers declared with block scope.
- A single translation unit can have at least 127 macro identifiers defined simultaneously.
- A macro definition can have at least 127 parameters and a macro invocation at least 31 arguments.
- A logical source line can have at least 4095 characters.
- A string literal or wide string literal can have at least 4095 characters (after string concatenation).
- An object can consist of at least 32767 bytes.
- A constant or variable can consist of at least 65535 bytes.
- #included files can have 15 nested levels.
- A switch statement can have 1023 case labels (exclusing any nested switch statements).
- A single struct or union can have at least 1023 members.
- A `switch` statement can have at least 1023 case labels (excluding any nested switch statements).
- A single `struct` or `union` can have at least 1023 members.
- A single enumeration can have at least 1024 enumeration constants.
- A single structure declaration can have at least 63 levels of nested structure or union definitions.

Implementation-Defined Behavior of TNS C

The ISO standard for C allows implementations to vary in specific instances. This subsection describes the implementation-defined behavior of TNS C. This subsection corresponds to Annex G.3 of the ISO C standard or Appendix F of the ANSI C standard.

G.3.1 Translation

Each nonempty sequence of white-space characters, other than newline, is retained.

The form of the diagnostic messages displayed by the TNS compiler is such that the source line is displayed first, followed by a line of the form: file name line diagnostic-type: diagnostic message. For example:

```
100   foo ();
**** \prune.$data.test.testc  100    Warning 95:
prototype function -declaration not in scope: “foo”
```

There are different classes of messages. They are:

ERROR

WARNING

The translator return status code for each class of message is:

0 = Normal, voluntary termination with no errors or warnings.

1 = Normal, voluntary termination with warning diagnostics.

2 = Abnormal, voluntary termination with warning diagnostics.

The level of diagnostic can be controlled. Control takes these form:

- For ERROR, the `ERRORS` pragma directs the compiler to terminate compilation if it detects more than a specified number of errors. If the `ERRORS` pragma is not used, the compiler completes a compilation regardless of the number of errors it diagnoses.
- For WARNING, the `WARN` pragma controls the generation of all or selected warning messages. The `WARN` pragma enables the compiler to generate all or a selected set of warning messages, and the `NOWARN` pragma disables the compiler from generating all or a selected set of warning messages. The compiler defaults to `WARN`, which enables all warning messages.

G.3.2 Environment

No library facilities are available to a freestanding program.

In a freestanding environment, program termination is:

- The program termination phase of execution begins when a program returns from the function main, calls the `exit()` library function, or calls the `terminate_program()` library function. In each case, the C library flushes all file buffers, closes all open files, and causes the process to complete with a certain completion code, depending on what caused the termination.
- When the process returns from main with no return value, the process stops with a completion code of 0, normal termination.
- When a program calls `exit()` or `terminate_program()`, the process completes with normal or abnormal termination depending upon the completion code assigned to the status or options parameters, respectively.

HPE TNS C allows the declaration of up to three parameters to the program's function main.

```
int main(int argc, char *argv[], char *env[])
```

Parameters	**Description**
`argc`	An integer value specifying the number of elements in the argument array `argv`
`argv`	The argument array. Each element (except for the last) points to a null-terminated string. `argv[0]` points to the fully qualified name of the executing program, the executing program name being that used in the RUN command. Each of the elements `argv[1]` through `argv[argc-1]` points to one command-line argument; `argv[argc]` has the pointer value NULL.
env	The environment array. Each element (except for the last) points to a null-terminated string containing the name and value of one environment parameter. The last element has the pointer value NULL.

When declaring parameters to the function main, note:

- The parameters to main are optional; there can be no parameters, `argc` and `argv`, or all three parameters. `env` alone is not allowed.
- The identifiers `argc`, `argv`, and `env` are simply the traditional names of the three parameters to main; any identifiers may be used.
- The elements of the environment array `env` point to strings of the form:

  `param-name=param-value`

  where

  param-name is the name of the environment parameter, and *param-value* is its value.

These are valid interactive devices:

- Asynchronous terminal
- Paired display and keyboard
- NonStop OS processes

G.3.3 Identifiers

No characters beyond 31 are significant in an identifier without external linkage. Beyond 6 characters, only 2 are significant in an identifier with external linkage. Case is significant in an identifier with external linkage.

G.3.4 Characters

There are no source and execution characters that are not explicitly specified by the ISO/ANSI C Standard.

These are the escape sequence values produced for the listed sequences:

Sequence	Value
\a	007 alert (bell)
\b	008 backspace
\f	012 form feed
\n	010 new line
\r	013 carriage return
\t	009 horizontal tab
\v	011 vertical tab

The HPE NonStop OS handles the shift states used for the encoding of multi-byte characters; therefore, it is not applicable to list them here.

There are 8 bits in a character in the execution character set.

There is a one-to-one mapping of members of the source character sets (in character and string literals) to members of the execution character set.

The value of an integer character constant, that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant, is -1.

When there is an integer character constant that contains more than one character or a wide character constant that contains more than one multi-byte character, the compiler concatenates the characters to form either a short or long integer, depending on the length of the constant.

The C-locale is used to convert multi-byte characters into corresponding wide characters for a wide character constant. A pointer to a copy of the locale is returned by the function `localeconv()`.

The equivalent type of a plain `char` is `unsigned char`.

G.3.5 Integers

This table describes the amount of storage and the range of various types of integers:

Designation	Size (bits)	Range
`char`	8	0 to 255
`signed char`	8	-128 to 127
`unsigned char`	8	0 to 255
`short`	16	-32,768 to 32,767
`signed short`	16	-32,768 to 32,767
`unsigned short`	16	0 to 65,535
`int`	32	-2,147,483,648 to 2,147,483,647
`signed int`	32	-2,147,483,648 to 2,147,483,647
`unsigned int`	32	0 to 4,294,967,295
`long`	32	-2,147,483,648 to 2,147,483,647
`signed long`	32	-2,147,483,648 to 2,147,483,647
`unsigned long`	32	0 to 4,294,967,295

The result of converting an integer to a shorter signed integer is undefined and may result in arithmetic overflow.

The result of converting an unsigned integer to a signed integer of equal length is undefined and may result in arithmetic overflow.

The result of bitwise operations on signed integers is the same as the result for unsigned integers, except for the right shift.

The result of a right shift of a negative-valued signed integral type is an arithmetic right shift for signed numbers and logical right shift for unsigned numbers.

The remainder on integer division has a positive sign (+).

G.3.6 Floating Point

This subsection applies only to Tandem floating-point format. The scaled value of a floating constant that is in the range of the representable value for its type is the larger representable value immediately adjacent to the nearest representable value.

This table describes the amount of storage and the range of various types of floating-point numbers.

Designation	Size(bits)	Range
float	32	8.63617e-78 to 1.15792e77
double	64	8.6361685550944446e-78 to 1.1579208923716189e77
long double	64	8.6361685550944446e-78 to 1.1579208923716189e77

When an integral number is converted to a floating-point number that cannot exactly represent the original value, the direction of truncation is up.

When a floating-point number is converted to a narrower floating-point number, the direction of truncation, or rounding is up.

G.3.7 Arrays and Pointers

An `unsigned long` integer is required to hold the maximum size of an array, that is the type of the size of operator `size_t`.

32 bits is the size of the integer required for a pointer to be converted to an integral type.

Casting a pointer to an integer returns: "Warning 32: invalid conversion specified." Casting an integer to a pointer works without error or warning.

An integer of type long is required to hold the difference between two pointers to members of the same array, `ptrdiff_t`.

G.3.8 Registers

The register storage class specifier has no effect on the actual storage of objects in registers.

G.3.9 Structures, Unions, Enumerations and Bit Fields

If a member of a union object is accessed using a member of a different type, the member is automatically cast to the type of the destination, as long as the conversion is valid. In other words, it works the same as conversions between variables that are not union members.

For the padding and alignment of members of structures, all fields are aligned on 16-bit boundaries, except characters that are aligned on byte boundaries.

The equivalent type for a plain `int` bit field is:

- `signed int` bit field
- `unsigned int` bit field

The order of allocation of bit fields within an int is high order to low order.

A bit field cannot straddle a storage-unit boundary.

The `int` integer type has been chosen to represent the values of an enumeration type.

G.3.10 Qualifiers

An access through an lvalue is needed to access a volatile-qualified type object.

G.3.11 Declarators

There is no practical limit to the maximum number of declarators that may modify an arithmetic, structure, or union type.

G.3.12 Statements

The maximum number of `case` values in a `switch` statement is MAX_INT.

G.3.13 Preprocessing Directives

The value of a single-character character constant in a constant expression that controls conditional inclusion does match the value of the same character constant in the execution character set. Such a character constant can have a negative value.

The method used for locating includable source files is:

- In the Guardian environment, for `#include`"*filename*" the current subvolume is searched or the subvolume search list specified by `pragma ssv0` through `ssv9` is searched, `ssv0` first and `ssv9` last. For `#include`*"subvol.filename"* the current volume is searched or the volume search list specified by `pragma ssv0` through `ssv9` is searched, `ssv0` first and `ssv9` last.
- In the Guardian environment for `#include "`*filename*`"`, if the include file does not have an absolute pathname, they will be searched first in the directory of the file with `#include` line, then in the directory named in `-I` option of `c89` utility and last in `/usr/include`.
- In the Guardian environment, for `#include <`*filename*`>`, the file is searched for in the compiler subvolume unless the search subvolumes (SSVs) are specified. If SSVs are specified, the SSVs are searched first and then the compiler subvolume.
- In the OSS environment on NonStop S-series systems, for `#include <`*filename*`>`, if the file does not have an absolute pathname, it is searched for first in the directory named in the `-I` option of the `c89` utility and then in the `/usr/include` directory.

Quoted names are supported for includable source files.

The mapping between delimited character sequences and an external source file name is:

- In the Guardian environment, it is a one-to-one mapping; except that when a file ends in ".h", the "." is removed.
- In the OSS environment on NonStop S-series systems, it is a one-to-one mapping.

For a description of the behavior of each recognized `#pragma` directive, see **Compiler Pragmas** on page 193.

The date and time of translation are always available.

G.3.14 Library Functions

OL is the null pointer constant to which the macro NULL expands.

For information on how to recognize the diagnostic that is printed by the assert() function and information on the termination behavior of the assert() function, see the assert() function in the *Guardian TNS C Library Calls Reference Manual*.

These characters are tested for by the isalnum(), isalpha(), iscntrl(), islower(), isprint(), and isupper() functions:

- isalnum() returns true for characters a–z, A–Z, 0–9
- isalpha() returns true for characters a–z, A–Z
- iscntrl() returns true for values 0–31, 127
- islower() returns true for characters a–z
- isprint() returns true for values 32–126
- isupper() returns true for characters A–Z

An undefined value is returned by the mathematics functions after a domain error, and errno is set to EDOM.

The mathematics functions set the integer expression errno to the value of the macro ERANGE on underflow range errors.

When the fmod() function has a second argument of zero, fmod() returns the value of the first argument (that is zero is treated as one).

Signals

For the set of signals for the Guardian signal() function and a description of the parameters and the usage of each signal, see the *Guardian TNS C Library Calls Reference Manual*.

For the set of signals for the OSS signal() function and a description of the parameters and the usage of each signal, see the signal(4) reference page online or in the *Open System Services System Calls Reference Manual*.

For each signal recognized by the signal() function, at program startup the handler SIG_DFL is registered for the all the signals by the C runtime.

The default handling is reset if a SIGILL signal is received by a handler specified to the signal() function.

Streams and Files

The last line of a text stream does not require a terminating newline character. The file is written with the characters requested.

The space characters that are written out to a text stream immediately before a newline character appear when the stream is read back in.

Zero null characters may be appended to data written to a binary stream.

The file position indicator of an append mode stream is initially positioned at the start of the file.

A write on a text stream does not cause the associated file to be truncated beyond that point.

Full buffering best describes the file buffering of the TNS C run-time.

A zero-length file can actually exist.

These are the rules for composing a valid file name:

- In the Guardian environment, the file name is formed as specified by the Guardian file-name format, which is composed of system name, volume name, subvolume name and file name, each separated by a period(.). Except for the file name, all the other fields can be omitted, then the default sub-volume is the present working subvolume.
- In the OSS environment on NonStop S-series systems, the newline file name must be a valid OSS filename.

The same file can be opened simultaneously multiple times, as long as it is not for writing.

The effect of the `remove()` function on an open file is that it returns a nonzero value and does not remove the file.

If a file with a new name already exists prior to a call to the `rename()` function, it returns a nonzero value without renaming the file.

The output for `%p` conversion in the `fprintf()` function formats the value of a pointer to a void argument using the format specified by the `x` conversion code.

The input for `%p` conversion in the fscanf() function matches an unsigned hexadecimal integer that uses the lowercase letters `a` through `f` to represent the digits 10 through 15. The corresponding *obj_ptr* must be a pointer to void. Consequently, the integer is interpreted as a pointer to void.

The hyphen character (`-`) is treated as just another character in the scan list if it is not the first or the last character in the scan list.

tmpfile()

An open temporary file is removed if the program terminates abnormally.

errno

The macro errno is set to 4009 by the `fgetpos()` or `ftell()` function on failure.

A call to the `perror()` function prints the textual error message corresponding to the value of the errno, optionally preceded by a specified string.

Memory

The behavior of the `calloc()`, `malloc()`, or `realloc()` function if the size requested is zero is:

- In the Guardian environment, `calloc()`, `malloc()`, and `realloc()` abort with runtime error 40.
- In the OSS environment on NonStop S-series systems, `calloc()`, `malloc()`, and `realloc()` return a non-NULL pointer. These pointers should not be dereferenced.

abort()

When the abort() function is called, all open and temporary files are closed. The temporary files are deleted.

exit()

The status returned by the `exit()` function if the value of the argument is other than zero, EXIT_SUCCESS, or EXIT_FAILURE is:

If the value is negative, the `exit()` function will return EXIT_FAILURE.

If the value is positive, the exit condition code is equal to status. See the *Guardian Procedure Calls Reference Manual*.

getenv()

The environment names are always in uppercase. These four run-time parameters are always present:

Parameter	Description
`STDIN`	Gives the name of the standard input file, `stdin`.
`STDOUT`	Gives the name of the standard output file, `stdout`.
`STDERR`	Gives the name of the standard error file, `stderr`.
`DEFAULTS`	Gives the default volume and subvolume names used to qualify partial file names.

To alter the environment list obtained by a call to the getenv() function, use the PARAM command, the syntax for which is:

```
PARAM param_name param_setting
```

param_name

is the run-time environment parameter name

param_setting

is the string that will be returned when getenv is called with *param_name* as its parameter.

system()

The string passed to the `system()` function is either a NULL string or an ASCII string that can be processed by the command processor.

strerror()

The format of the error message returned by the `strerror()` function is an ASCII string.

errno Values and Corresponding Messages lists the contents of the error message strings returned by a call to the strerror() function:

Table 71: errno Values and Corresponding Messages

errno	**Message**
EPERM	Not owner, permission denied
ENOENT	No such file or directory
ESRCH	No such process or table entry
EINTR	Interrupted system call
EIO	I/O error
ENXIO	No such device or address

Table Continued

errno	Message
E2BIG	Argument list too long
ENOEXEC	Exec format error
EBADF	Bad file descriptor
ECHILD	No children
EAGAIN	No more processes
ENOMEM	Insufficient user memory
EACCES	Permission denied
EFAULT	Bad address
EBUSY	Mount device busy
EEXIST	File already exists
EXDEV	Cross-device link
ENODEV	No such device
ENOTDIR	Not a directory
EISDIR	Is a directory
EINVAL	Invalid function argument
ENFILE	File table overflow
EMFILE	Maximum number of files already open
ENOTTY	Not a typewriter
EFBIG	File too large
ENOSPC	No space left on device
ESPIPE	Illegal seek
EROFS	Read only file system
EMLINK	Too many links
EPIPE	Broken pipe or no reader on socket
EDOM	Argument out of function's domain

Table Continued

errno	Message
ERANGE	Value out of range
EDEADLK	Deadlock condition
ENOLCK	No record locks available
ENODATA	No data sent or received
ENOSYS	Function not implemented
EWOULDBLOCK	Operation would block
EINPROGRESS	Operation now in progress
EALREADY	Operation already in progress
ENOTSOCK	Socket operation on non-socket
EDESTADDRREQ	Destination address required
EMSGSIZE	Message too long
EPROTOTYPE	Protocol wrong type for socket
ENOPROTOOPT	Protocol not available
EPROTONOSUPPORT	Protocol not supported
ESOCKTNOSUPPORT	Socket type not supported
EOPNOTSUPP	Operation not supported on socket
EPFNOSUPPORT	Protocol family not supported
EAFNOSUPPORT	Address family not supported
EADDRINUSE	Address already in use
EADDRNOTAVAIL	Can't assign requested address
ENETDOWN	Network is down
ENETUNREACH	Network is unreachable
ENETRESET	Network dropped connection on reset
ECONNABORTED	Software caused connection abort
ECONNRESET	Connection reset by remote host

Table Continued

errno	Message
ENOBUFS	No buffer space available
EISCONN	Socket is already connected
ENOTCONN	Socket is not connected
ESHUTDOWN	Can't send after socket shutdown
ETIMEDOUT	Connection timed out
ECONNREFUSED	Connection refused
EHOSTDOWN	Host is down
EHOSTUNREACH	No route to host
ENAMETOOLONG	File name too long
ENOTEMPTY	Directory not empty
EHAVEOOB	Out-of-band data available
EBADSYS	Invalid socket call
EBADFILE	File type not supported
EBADCF	Not a C file
ENOIMEM	Insufficient internal memory
EBADDATA	Invalid data in buffer
ENOREPLY	No reply in buffer
EPARTIAL	Partial buffer received
ESPIERR	Interface error from SP I
EVERSION	Version mismatch
EXDRDECODE	XDR encoding error
EXDRENCODE	XDR decoding error

G.4 Locale Behavior

The local time zone and daylight-saving time depend on the system location.

The era for the clock function "Locale-specific Behavior" is January 1, 1970 GMT.

No characters have been added to the execution set required by the ISO/ANSI C standard.

The direction of printing is left to right, and top to bottom.

The decimal point character is a period (.).

G.5 Common Extensions

There are no common extensions to the formats for time and date.

Multibyte Characters and Wide Characters

Multibyte characters and wide characters support Asian alphabets that often contain a very large number of characters. The Guardian TNS C run-time library functions, except for the `strcoll()` and `strxfrm()` functions, support these character sets: Tandem Kanji, Chinese Big 5, Chinese PC, Hangul and KSC5601.

Discussion of multibyte characters applies only to the Guardian environment. For more details on multibyte characters in the Open System Services (OSS) environment, see the *Software Internationalization Manual*.

The D30 and later Guardian C run-time library functions `mblen()`, `mbtoc()`, `mbtowcs()`, `wctomb()`, and `wctombs()` do not support multibyte characters for programs that use the 32-bit (or wide) data model as described in this section. Guardian programs that use the 32-bit data model must use the Guardian system procedures that support multibyte characters instead. For more details, see the *Guardian Programmer's Guide*.

The default character set supported by a system is configured at system installation time and cannot be changed during program execution. The Guardian procedure MBCS_DEFAULTCHARSET_ returns the identifier of the default character set. The *Guardian Procedure Calls Reference Manual* describes this system procedure in detail.

The internal representation of the characters of these languages is HPE internal and might not conform to any ISO standard. HPE can choose to change this internal representation at any time.

Multibyte Characters

- The basic difficulty in an Asian environment is the huge number of ideograms that are needed for I/0, for example Chinese characters. To work within the constraints of usual computer architectures, these ideograms are encoded as sequences of bytes. The associated operating systems, application programs, and terminals understand these byte sequences as individual ideograms. Moreover, all these encodings allow intermixing of regular single-byte C characters with the ideogram byte sequences.
- The term “multibyte character” denotes a byte sequence that encodes an ideogram. The byte sequence contains one or more codes where each code can be represented in a C character data type: char, signed char, or unsigned char. All multibyte characters are members of the so-called extended character set. A regular single-byte C character is just a special case of a multibyte sequence where the sequence has a length of one.

Wide Characters

Some of the inconvenience of handling multibyte characters is eliminated if all characters are of a uniform number of bytes or bits. A 16-bit integer value is used to represent all members because there can be thousands or tens of thousands of ideograms in an Asian character set.

Wide characters are integers of type `wchar_t`, defined in the headers `stddef.h` and `stdlib.h` as:

```
typedef unsigned short wchar_t;
```

Such an integer can represent distinct codes for each of the characters in the extended character set. The codes for the basic C character set have the same values as their single-character forms.

Relationship Between Multibyte and Wide Characters

- Multibyte characters are convenient for communicating between the program and the outside world.
- Wide characters are convenient for manipulating text within a program.
- The fixed size of wide characters simplifies handling both individual characters and arrays of characters.

`MB_CUR_MAX` Macro

The `MB_CUR_MAX` macro specifies the maximum number of bytes used in representing a multibyte character in the current locale (category `LC_CTYPE`). The `MB_CUR_MAX` macro is defined in the header stdlibh as:

```
#define MB_CUR_MAX  2
```

Conversion Functions

The five run-time library functions that manage multibyte characters and wide characters are:

Function	Description
`mblen()`	Determines the length of a multibyte character.
`mbtowc()`	Converts a multibyte character to a wide character.
`wctomb()`	Converts a wide character to a multibyte character.
`mbstowcs()`	Converts a string of multibyte characters to a string of wide characters.
`wcstombs()`	Converts a string of wide characters to a string of multibyte characters.

Alignment Issues

TNS C and C++ considers objects of integral types to exist only on word boundaries. Consequently, it is invalid to use an odd-byte address to access such an object. On TNS systems, if an integral type extended pointer contains an odd-byte address, the system ignores the last bit of the address and accesses the object in the even address one byte below. On TNS/R, TNS/E, and TNS/X systems, the results of using an integral type extended pointer containing an odd-byte address are undefined. The code might continue executing or trap. Therefore, it is important for you to ensure that all extended pointers contain addresses that are even except for pointers to `char`. Extended pointers are those of type long int or those of type `int` with the 32-bit (wide) data model in effect, in which case an `int` is represented by 32 bits.

TNS systems are word-aligned. In this example, the size of `str` differs depending on whether this code is executed on a word-aligned or byte aligned machine. On a word aligned machine `str` is allocated 6 bytes of storage. On a byte-aligned machine, `str` is allocated 5 bytes of storage. On the TNS system, `str` is allocated 6 bytes of storage.

```
Struct tag{
   char c[3];
   int x;
} str;
```

TNS C++ Implementation-Defined Behavior

HPE Specific Features for OSS and Guardian Environments

The HPE specific features discussed in this subsection are applicable to the Guardian environment on all current systems and to the OSS environment on NonStop S-series systems.

Length of Identifiers

Cfront generates identifiers up to a maximum of 230 characters except for global variable names which can be up to a maximum of 127 characters.

For identifiers longer than 230, characters 229 and 230 represent a hashing of all characters beyond character 228. For global variable names longer than 127, characters 126 and 127 represent a hashing of all characters beyond character 125. This rule applies to identifiers and global variable names that you provide and that are generated by Cfront as a result of name encoding.

Length of String Literals

Cfront supports a maximum string literal size of 4096 characters, which is compatible with the string literal size supported by the C compiler.

Data Types

HPE C++ supports the standard data types supported in the HPE C programming language, including data types `signed char` and `long long`, a 64 bit-integer.

The data type `signed char` is an HPE extension to Cfront. The data type `signed char` can be used to distinguish a specific instance of an overloaded member function.

The data type `long long` is also an HPE extension to Cfront. The data type `long long` is recognized as a predefined data type. Type matching occurs for initialization, expressions, and argument lists. Variables of type `long long` are allowed in all contexts that HPE C allows them. Constants of type `long long` are recognized when the decimal number is followed by the modifier `LL` or `ll`. Variables and constants of type `long long` are allowed in classes and templates.

Type Qualifiers

HPE C++ recognizes the type qualifier `volatile`, but ignores it. You cannot use `volatile` to distinguish specific instances of overloaded member functions.

Templates (Parameterized Types)

HPE C++ provides language support for templates. All templates must be instantiated at compile time. This requires that template bodies be available at compile time. There is no support for bind-time instantiation.

To ensure minimal change by users of future releases of C++, these organization for programs using templates is recommended:

1. Put the declaration of template T in the header file TH.
2. Put template T function bodies in file TC.
3. Put `#include "TC"` as the last line of header file TH so as to include the function body file TC.

This organization ensures that any file including TH will also include TC. If a future release of HPE C++ supports bind-time instantiation, the single line `#include "TC"` can be removed.

Class Libraries

The iostream class library described in *The Annotated C++ Reference Manual* is available for use with C+ +.

In the Guardian environment, Cprep truncates the UNIX named iostream header file from `iostream.h` to `iostreah` to comply with the Guardian file naming conventions. However, for portability, you can use `iostream.h`.

Interfacing to the Standard C Run-Time Library

To declare a standard C run-time library function, use the `#include` preprocessor directive to include the name of the header file that contains the function prototype for the particular function.

For example:

```
#include <stdio.h>
```

Interfacing to User-Defined C Libraries

If you have user-defined C function prototypes defined in a header file, designate the header file as having C linkage. Therefore, use the extern C construct. For example:

```
extern "C" {
#include "myclib.h"
}
```

Interfacing to User-Defined C Functions

To declare a user-defined C function, you can use the C++ extern C construct, which is a standard C++ language feature. Here is an example of using the extern C construct to declare a user-defined C function:

```
extern "C" void f(int);
```

If you want to declare several user-defined C functions at the same time, you can use braces. For example,

```
extern "C" {
     void f(int);
    int g(char);
}
```

The extern C construct states that the specified functions are C functions so that there is no name encoding. If you do not include the user defined C function in the extern C construct, you will get errors when Binder tries to bind a module that contains these functions.

In C++, function names are encoded or mangled to produce unique internal names, which incorporate function argument types. In C there is no name encoding. Therefore, if you want to call a C function, its name will not be the same as that produced by the C++ translator and used by the Binder. To indicate that C linkage applies, rather than C++, you must use the extern C construct.

Interfacing to NonStop SQL/MP

Cfront does not support embedded SQL. Embedded SQL support is provided by binding in C modules containing the desired SQL statements. Keep all SQL statements in one or more separately compiled C modules and bind these C modules into the executable C++ object file with Binder.

HPE Specific Features for the Guardian Environment

The HPE specific features discussed in this subsection are applicable to only the Guardian environment.

Mixed-Language Programming

Mixed-language programming in the HPE environment comprises invoking a C, COBOL85, FORTRAN, Pascal, or TAL routine from your C++ program, or calling a C++ function from one of these other languages. A function declaration must always precede any invocation of that function.

For information on interfacing to other languages, refer to the discussion on mixed-language programming in **Mixed-Language Programming for TNS Programs** on page 113, and in **Mixed-Language Programming for TNS/R, TNS/E, and TNS/X Native Programs** on page 141.

To call a C++ function from another language, call the C++ function as if it were a C function but use the encoded name form. Get this encoded name form from the C source file that is created by Cfront.

If you want to call a function written in another language from a C or C++ program, your program must have a C main function.

You do not need to put the extern C construct around `#include <cextdecs>` because the extern C construct has already been added to the `<cextdecs>` header file for use by C++.

System-Level Programming

C++ supports system-level programming. System-level programming refers to the ability to write C++ functions that reside in system code, system library, or user library. Refer to **System-Level Programming** on page 166.

Differences Between OSS and Guardian Environments

The OSS and Guardian versions of Cfront each default to generate code that is destined to run in their own environment. This subsection discusses how to target code to run in the other environment and describes the differences that exist between an executable C++ program file in the two environments.

Pragma SYSTYPE

The pragma SYSTYPE specifies whether a C++ program is targeted to run in the OSS or the Guardian environment.

The Guardian version of Cfront defaults to generating code that is targeted for the Guardian environment. However, when the pragma SYSTYPE OSS is specified, the Guardian version of Cfront generates code targeted for the OSS environment.

The OSS version of Cfront defaults to generating code that is targeted for the OSS environment. However, when the pragma SYSTYPE GUARDIAN is specified, the OSS version of Cfront generates code targeted for the Guardian environment.

Data Models

C++ programs that are generated with the OSS version of Cfront always use the 32-bit data model. Under the 32-bit data model the size of type int is 32 bits. C++ programs that are generated with the Guardian version of Cfront also default to the 32-bit data model. Therefore, the data models will be compatible unless you specify the NOWIDE pragma when compiling the Guardian version. The NOWIDE pragma specifies the 16-bit data model, where the size of type int is 16 bits.

If 16-bit and 32-bit data are both required, you might want to avoid type int and use types short and long instead.

Run-Time Libraries

The OSS environment has only one C++ run-time library, `libC.a`.

The Guardian environment has two run-time libraries. LIBCA is used with the 32-bit data model. LIBLA is used with the 16-bit data model.

Applicable Pragmas

In the OSS environment, the C compiler ignores these pragmas: ANSISTREAMS, RUNNABLE, SEARCH, SSV, STDFILES, and XVAR.

For additional information on writing programs for the OSS environment, see **Compiling, Binding, and Accelerating TNS C Programs** on page 336, and **Compiling, Binding, and Accelerating TNS C++ Programs** on page 351.

ASCII Character Set

Overview

The two tables of the ASCII character set contained in this appendix use these column headings:

Ord.	Character’s ordinal number in the ASCII character set
Octal	Character’s octal representation (with left and right bytes)
Hex.	Character’s hexadecimal representation
Dec.	Character’s decimal representation
Char	Character code or character itself (such as “NUL” or “A”)
Meaning	Meaning of character code (such as “Null” or “Uppercase A”)

ASCII Character Set in Numeric Order is in numeric order, and **ASCII Character Set in Alphabetic Order is in alphabetic order**.

ASCII Character Set in Numeric Order

ASCII Character Set in Numeric Order presents the ASCII character set in numeric order; that is, these columns are in numeric order:

- Ord. (the character’s ordinal number in the ASCII character set)
- Octal (the character’s octal representation)
- Hex. (the character’s hexadecimal representation)
- Dec. (the character’s decimal representation)

If you know one of the above values for the character you want to look up, use **ASCII Character Set in Numeric Order**. If you know only the character code or the character itself (such as “NUL” or “A”), use **ASCII Character Set in Alphabetic Order is in alphabetic order** instead.

Table 72: ASCII Character Set in Numeric Order

	Octal					
Ord.	**Left**	**Right**	**Hex.**	**Dec.**	**Char.**	**Meaning**
1	000000	000000	00	0	NUL	Null
2	000400	000001	01	1	SOH	Start of heading
3	001000	000002	02	2	STX	Start of text

Table Continued

	Octal					
Ord.	**Left**	**Right**	**Hex.**	**Dec.**	**Char.**	**Meaning**
4	001400	000003	03	3	ETX	End of text
5	002000	000004	04	4	EOT	End of transmission
6	002400	000005	05	5	ENQ	Enquiry
7	003000	000006	06	6	ACK	Acknowledge
8	003400	000007	07	7	BEL	Bell
9	004000	000010	08	8	BS	Backspace
10	004400	000011	09	9	HT	Horizontal tabulation
11	005000	000012	0A	10	LF	Line feed
12	005400	000013	0B	11	VT	Vertical tabulation
13	006000	000014	0C	12	FF	Form feed
14	006400	000015	0D	13	CR	Carriage return
15	007000	000016	0E	14	SO	Shift out
16	007400	000017	0F	15	SI	Shift in
17	010000	000020	10	16	DLE	Data link escape
18	010400	000021	11	17	DC1	Device control 1
19	011000	000022	12	18	DC2	Device control 2
20	011400	000023	13	19	DC3	Device control 3
21	012000	000024	14	20	DC4	Device control 4
22	012400	000025	15	21	NAK	Negative acknowledge
23	013000	000026	16	22	SYN	Synchronous idle
24	013400	000027	17	23	ETB	End of transmission block
25	014000	000030	18	24	CAN	Cancel
26	014400	000031	19	25	EM	End of medium
27	015000	000032	1A	26	SUB	Substitute

Table Continued

	Octal					
Ord.	Left	Right	Hex.	Dec.	Char.	Meaning
28	015400	000033	1B	27	ESC	Escape
29	016000	016000	1C	28	FS	File separator
30	016400	000035	1D	29	GS	Group separator
31	017000	000036	1E	30	RS	Record separator
32	017400	000037	1F	31	US	Unit separator
33	020000	000040	20	32	SP	Space
34	020400	000041	21	33	!	Exclamation point
35	021000	000042	22	34	"	Quotation mark
36	021400	000043	23	35	#	Number sign
37	022000	000044	24	36	$	Dollar sign
38	022400	000045	25	37	%	Percent sign
39	023000	000046	26	38	&	Ampersand
40	023400	000047	27	39	'	Apostrophe
41	024000	000050	28	40	(	Opening parenthesis
42	024400	000051	29	41	)	Closing parenthesis
43	025000	000052	2A	42	*	Asterisk
44	025400	000053	2B	43	+	Plus
45	026000	000054	2C	44	,	Comma
46	026400	000055	2D	45	-	Hyphen (minus)
47	027000	000056	2E	46	.	Period (decimal point)
48	027400	000057	2F	47	/	Slash
49	030000	000060	30	48	0	Zero
50	030400	000061	31	49	1	One
51	031000	000062	32	50	2	Two

Table Continued

	Octal					
Ord.	**Left**	**Right**	**Hex.**	**Dec.**	**Char.**	**Meaning**
52	031400	000063	33	51	3	Three
53	032000	000064	34	52	4	Four
54	032400	000065	35	53	5	Five
55	033000	000066	36	54	6	Six
56	033400	000067	37	55	7	Seven
57	034000	000070	38	56	8	Eight
58	034400	000071	39	57	9	Nine
59	035000	000072	3A	58	:	Colon
60	035400	000073	3B	59	;	Semicolon
61	036000	000074	3C	60	<	Less than
62	036400	000075	3D	61	=	Equals
63	037000	000076	3E	62	>	Greater than
64	037400	000077	3F	63	?	Question mark
65	040000	000100	40	64	@	Commercial at sign
66	040400	000101	41	65	A	Uppercase A
67	041000	000102	42	66	B	Uppercase B
68	041400	000103	43	67	C	Uppercase C
69	042000	000104	44	68	D	Uppercase D
70	042400	000105	45	69	E	Uppercase E
71	043000	000106	46	70	F	Uppercase F
72	043400	000107	47	71	G	Uppercase G
73	044000	000110	48	72	H	Uppercase H
74	044400	000111	49	73	I	Uppercase I
75	045000	000112	4A	74	J	Uppercase J

Table Continued

	Octal					
Ord.	Left	Right	Hex.	Dec.	Char.	Meaning
76	045400	000113	4B	75	K	Uppercase K
77	046000	000114	4C	76	L	Uppercase L
78	046400	000115	4D	77	M	Uppercase M
79	047000	000116	4E	78	N	Uppercase N
80	047400	000117	4F	79	O	Uppercase O
81	050000	000120	50	80	P	Uppercase P
82	050400	000121	51	81	Q	Uppercase Q
83	051000	000122	52	82	R	Uppercase R
84	051400	000123	53	83	S	Uppercase S
85	052000	000124	54	84	T	Uppercase T
86	052400	000125	55	85	U	Uppercase U
87	053000	000126	56	86	V	Uppercase V
88	053400	000127	57	87	W	Uppercase W
89	054000	000130	58	88	X	Uppercase X
90	054400	000131	59	89	Y	Uppercase Y
91	055000	000132	5A	90	Z	Uppercase Z
92	055400	000133	5B	91	[	Opening bracket
93	056000	000134	5C	92	\	Backslash
94	056400	000135	5D	93	]	Closing bracket
95	057000	000136	5E	94	^	Circumflex
96	057400	000137	5F	95	_	Underscore
97	060000	000140	60	96	`	Grave accent
98	060400	000141	61	97	a	Lowercase a
99	061000	000142	62	98	b	Lowercase b

Table Continued

	Octal					
Ord.	Left	Right	Hex.	Dec.	Char.	Meaning
100	061400	000143	63	99	c	Lowercase c
101	062000	000144	64	100	d	Lowercase d
102	062400	000145	65	101	e	Lowercase e
103	063000	000146	66	102	f	Lowercase f
104	063400	000147	67	103	g	Lowercase g
105	064000	000150	68	104	h	Lowercase h
106	064400	000151	69	105	i	Lowercase i
107	065000	000152	6A	106	j	Lowercase j
108	065400	000153	6B	107	k	Lowercase k
109	066000	000154	6C	108	l	Lowercase l
110	066400	000155	6D	109	m	Lowercase m
111	067000	000156	6E	110	n	Lowercase n
112	067400	000157	6F	111	o	Lowercase o
113	070000	000160	70	112	p	Lowercase p
114	070400	000161	71	113	q	Lowercase q
115	071000	000162	72	114	r	Lowercase r
116	071400	000163	73	115	s	Lowercase s
117	072000	000164	74	116	t	Lowercase t
118	072400	000165	75	117	u	Lowercase u
119	073000	000166	76	118	v	Lowercase v
120	073400	000167	77	119	w	Lowercase w
121	074000	000170	78	120	x	Lowercase x
122	074400	000171	79	121	y	Lowercase y
123	075000	000172	7A	122	z	Lowercase z

Table Continued

	Octal					
Ord.	Left	Right	Hex.	Dec.	Char.	Meaning
124	075400	000173	7B	123	{	Opening brace
125	076000	000174	7C	124	|	Vertical line
126	076400	000175	7D	125	}	Closing brace
127	077000	000176	7E	126	~	Tilde
128	077400	000177	7F	127	DEL	Delete

ASCII Character Set in Alphabetic Order

ASCII Character Set in Alphabetic Order is in alphabetic order presents the ASCII character set in alphabetic order—that is, alphabetic character codes (in the column labeled “Char.”) are in alphabetic order.

Table 73: ASCII Character Set in Alphabetic Order

			Octal			
Char	Meaning	Ord	Left	Right	Hex.	Dec.
^	Circumflex	95	057000	000136	5E	94
~	Tilde	127	077000	000176	7E	126
!	Exclamation point	34	020400	000041	21	33
"	Quotation mark	35	021000	000042	22	34
#	Number sign	36	021400	000043	23	35
$	Dollar sign	37	022000	000044	24	36
%	Percent sign	38	022400	000045	25	37
&	Ampersand	39	023000	000046	26	38
'	Apostrophe	40	023400	000047	27	39
(	Opening parenthesis	41	024000	000050	28	40
)	Closing parenthesis	42	024400	000051	29	41
*	Asterisk	43	025000	000052	2A	42
+	Plus	44	025400	000053	2B	43

Table Continued

			Octal			
Char	**Meaning**	**Ord**	**Left**	**Right**	**Hex.**	**Dec.**
,	Comma	45	026000	000054	2C	44
-	Hyphen (minus)	46	026400	000055	2D	45
.	Period (decimal point)	47	027000	000056	2E	46
/	Slash	48	027400	000057	2F	47
0	Zero	49	030000	000060	30	48
1	One	50	030400	000061	31	49
2	Two	51	031000	000062	32	50
3	Three	52	031400	000063	33	51
4	Four	53	032000	000064	34	52
5	Five	54	032400	000065	35	53
6	Six	55	033000	000066	36	54
7	Seven	56	033400	000067	37	55
8	Eight	57	034000	000070	38	56
9	Nine	58	034400	000071	39	57
:	Colon	59	035000	000072	3A	58
;	Semicolon	60	035400	000073	3B	59
<	Less than	61	036000	000074	3C	60
=	Equals	62	036400	000075	3D	61
>	Greater than	63	037000	000076	3E	62
?	Question mark	64	037400	000077	3F	63
@	Commercial at sign	65	040000	000100	40	64
[	Opening bracket	92	055400	000133	5B	91
\	Backslash	93	056000	000134	5C	92
]	Closing bracket	94	056400	000135	5D	93

Table Continued

			Octal			
Char	Meaning	Ord	Left	Right	Hex.	Dec.
_	Underscore	96	057400	000137	5F	95
`	Grave accent	97	060000	000140	60	96
A	Uppercase A	66	040400	000101	41	65
a	Lowercase a	98	060400	000141	61	97
ACK	Acknowledge	7	003000	000006	06	6
B	Uppercase B	67	041000	000102	42	66
b	Lowercase b	99	061000	000142	62	98
BEL	Bell	8	003400	000007	07	7
BS	Backspace	9	004000	000010	08	8
C	Uppercase C	68	041400	000103	43	67
c	Lowercase c	100	061400	000143	63	99
CAN	Cancel	25	014000	000030	18	24
CR	Carriage return	14	006400	000015	0D	13
D	Uppercase D	69	042000	000104	44	68
d	Lowercase d	101	062000	000144	64	100
DC1	Device control 1	18	010400	000021	11	17
DC2	Device control 2	19	011000	000022	12	18
DC3	Device control 3	20	011400	000023	13	19
DC4	Device control 4	21	012000	000024	14	20
DEL	Delete	128	077400	000177	7F	127
DLE	Data link escape	17	010000	000020	10	16
E	Uppercase E	70	042400	000105	45	69
e	Lowercase e	102	062400	000145	65	101
EM	End of medium	26	014400	000031	19	25

Table Continued

			Octal			
Char	**Meaning**	**Ord**	**Left**	**Right**	**Hex.**	**Dec.**
ENQ	Enquiry	6	002400	000005	05	5
EOT	End of transmission	5	002000	000004	04	4
ESC	Escape	28	015400	000033	1B	27
ETB	End of transmission block	24	013400	000027	17	23
ETX	End of text	4	001400	000003	03	3
F	Uppercase F	71	043000	000106	46	70
f	Lowercase f	103	063000	000146	66	102
FF	Form feed	13	006000	000014	0C	12
FS	File separator	29	016000	016000	1C	28
G	Uppercase G	72	043400	000107	47	71
g	Lowercase g	104	063400	000147	67	103
GS	Group separator	30	016400	000035	1D	29
H	Uppercase H	73	044000	000110	48	72
h	Lowercase h	105	064000	000150	68	104
HT	Horizontal tabulation	10	004400	000011	09	9
I	Uppercase I	74	044400	000111	49	73
i	Lowercase i	106	064400	000151	69	105
J	Uppercase J	75	045000	000112	4A	74
j	Lowercase j	107	065000	000152	6A	106
K	Uppercase K	76	045400	000113	4B	75
k	Lowercase k	108	065400	000153	6B	107
L	Uppercase L	77	046000	000114	4C	76
l	Lowercase l	109	066000	000154	6C	108
LF	Line feed	11	005000	000012	0A	10

Table Continued

			Octal			
Char	**Meaning**	**Ord**	**Left**	**Right**	**Hex.**	**Dec.**
M	Uppercase M	78	046400	000115	4D	77
m	Lowercase m	110	066400	000155	6D	109
N	Uppercase N	79	047000	000116	4E	78
n	Lowercase n	111	067000	000156	6E	110
NAK	Negative acknowledge	22	012400	000025	15	21
NUL	Null	1	000000	000000	00	0
O	Uppercase O	80	047400	000117	4F	79
o	Lowercase o	112	067400	000157	6F	111
P	Uppercase P	81	050000	000120	50	80
p	Lowercase p	113	070000	000160	70	112
Q	Uppercase Q	82	050400	000121	51	81
q	Lowercase q	114	070400	000161	71	113
R	Uppercase R	83	051000	000122	52	82
r	Lowercase r	115	071000	000162	72	114
RS	Record separator	31	017000	000036	1E	30
S	Uppercase S	84	051400	000123	53	83
s	Lowercase s	116	071400	000163	73	115
SI	Shift in	16	007400	000017	0F	15
SO	Shift out	15	007000	000016	0E	14
SOH	Start of heading	2	000400	000001	01	1
SP	Space	33	020000	000040	20	32
STX	Start of text	3	001000	000002	02	2
SUB	Substitute	27	015000	000032	1A	26
SYN	Synchronous idle	23	013000	000026	16	22

Table Continued

			Octal			
Char	Meaning	Ord	Left	Right	Hex.	Dec.
T	Uppercase T	85	052000	000124	54	84
t	Lowercase t	117	072000	000164	74	116
U	Uppercase U	86	052400	000125	55	85
u	Lowercase u	118	072400	000165	75	117
US	Unit separator	32	017400	000037	1F	31
V	Uppercase V	87	053000	000126	56	86
v	Lowercase v	119	073000	000166	76	118
VT	Vertical tabulation	12	005400	000013	0B	11
W	Uppercase W	88	053400	000127	57	87
w	Lowercase w	120	073400	000167	77	119
X	Uppercase X	89	054000	000130	58	88
x	Lowercase x	121	074000	000170	78	120
Y	Uppercase Y	90	054400	000131	59	89
y	Lowercase y	122	074400	000171	79	121
Z	Uppercase Z	91	055000	000132	5A	90
z	Lowercase z	123	075000	000172	7A	122
{	Opening brace	124	075400	000173	7B	123
|	Vertical line	125	076000	000174	7C	124
}	Closing brace	126	076400	000175	7D	125

Data Type Correspondence

The table **Integer Types, Part 1** contains the return value size generated by HP language compilers for each data type. Use this information when you need to specify values with the Accelerator ReturnValSize option. These tables are also useful if your programs use data from files created by programs in another language, or your programs pass parameters to programs written in callable languages.

Note that the return value sizes given in these tables do not correspond to the storage size of SQL data types. For a complete list of SQL data type correspondence, see:

- *SQL/MP Programming Manual for C*
- *SQL/MX Programming Manual for C and COBOL*

.

NOTE: COBOL includes COBOL 74, COBOL85, and SCREEN COBOL unless otherwise noted.

If you are using the Data Definition Language (DDL) utility to describe your files, see the *Data Definition Language (DDL) Reference Manual* for more information.

Table 74: Integer Types, Part 1

	8-Bit Integer	**16-Bit Integer**	**32-Bit Integer**
C-series BASIC	STRING	INTINT(16)	INT(32)
C and C++	char[1] unsigned char signed char	int in the 16-bit data model short unsigned	int in the 32-bit or wide data model long unsigned long
COBOL	AlphabeticNumeric DISPLAYAlphanumeric-EditedAlphanumericNumeric-Edited	PIC S9(n) COMP or PIC 9(n) COMP without P or V, 1 < n < 4 Index Data Item[2] NATIVE-2[3]	PIC S9(n) COMP or PIC 9(n) COMP without P or V, 5 < n < 9 Index Data Item[2] NATIVE-4[3]
FORTRAN	--	INTEGER[4] INTEGER*2	INTEGER*4
D-series Pascal	BYTE Enumeration, unpacked, 256 members Subrange, unpacked, n…m, 0 < n and m < 255	INTEGER INT16 CARDINAL[1] BYTE or CHAR value parameter Enumeration, unpacked, > 256 members Subrange, unpacked, n…m, -32768 < n and m < 32767, but at least n or m outside 0…255 range	LONGINT INT32 Subrange, unpacked n…m, –2147483648 < n and m < 2147483647, but at least n or m outside -32768…32767 range

Table Continued

	8-Bit Integer	16-Bit Integer	32-Bit Integer
SQL	CHAR	NUMERIC(1)… NUMERIC(4) PIC 9(1) COMP… PIC 9(4) COMPSMALLINT	NUMERIC(5)… NUMERIC(9) PIC 9(5) COMP …PIC 9(9) COMPINTEGER

TAL and pTAL	STRING
	UNSIGNED(8)
		INT
		INT(16)
		UNSIGNED(16)
			INT(32)

Return Value Size (Words)	1	1	2

[1] Unsigned Integer

[2] Index Data Item is a 16-bit integer in COBOL 74 and a 32-bit integer in HP COBOL85.

[3] HP COBOL85 only.

[4] INTEGER is normally equivalent to INTEGER*2. The INTEGER*4 and INTEGER*8 compiler directives redefine INTEGER.

Table 75: Integer Types, Part 2

	64-Bit Integer	Bit Integer of 1 to 31 Bits	Decimal Integer

C-series BASIC	INT(64)
	FIXED(0)
		--	--

C and C++	long long unsigned long long	--	--
COBOL	PIC S9(n) COMP or PIC 9(n) COMP without P or V, 10 < n < 18NATIVE-8[1]	--	Numeric DISPLAY
FORTRAN	INTEGER*8	--	--
D-series Pascal	INT64	UNSIGNED(n), 1 < n < 16 INT(n), 1 < n < 16	DECIMAL
SQL	NUMERIC(10)… NUMERIC(18) PIC 9(10) COMP… PIC 9(18) COMP LARGEINT	--	DECIMAL (n,s)PIC 9(n) DISPLAY

Table Continued

	64-Bit Integer	Bit Integer of 1 to 31 Bits	Decimal Integer
TAL and pTAL	FIXED(0), INT(64)	UNSIGNED(n), 1 < n < 31	--
Return Value Size (Words)	4	1 or 2 in TAL, 1 in other languages	1 or 2, depends on declared pointer size

[1] HP TNS COBOL only.

Table 76: Floating, Fixed, and Complex Types

	32-Bit Floating	64-Bit Floating	64-Bit Fixed Point	64-Bit Complex
C-series BASIC	REAL	REAL(64)	FIXED(s), 0 < s < 18	--
C and C++	float	double	--	--
COBOL	--	--	PIC S9(n–s)v9(s) COMP or PIC 9(n–s)v9(s) COMP, 10 < n < 18	--
FORTRAN	REAL	DOUBLE PRECISION	--	COMPLEX
D-series Pascal	REAL	LONGREAL	--	--
SQL	--	--	NUMERIC (n,s) PIC 9(n-s)v9(s) COMP	--
TAL and pTAL	REALREAL(32)	REAL(64)	FIXED(s), -19 < s < 19	--
Return Value Size (Words)	2	4	4	4

Table 77: Character Types

	Character	Character String	Variable-Length Character String
C-series BASIC	STRING	STRING	--

C and C++	signed char
	unsigned char
		pointer to char	struct {
			int len;
			char val [n]
			};

Table Continued

	Character	Character String	Variable-Length Character String
COBOL	Alphabetic Numeric DISPLAY Alphanumeric-Edited Alphanumeric Numeric-Edited	Alphabetic Numeric DISPLAY Alphanumeric-Edited Alphanumeric Numeric-Edited	01 name. 03 len USAGE IS NATIVE-2[1] 03 val PIC X(n).
FORTRAN	CHARACTER	CHARACTER array CHARACTER*n	--
D-series Pascal	CHAR or BYTE value parameterEnumeration, unpacked, < 256 membersSubrange, unpacked n…m, 0 < n and m < 255	PACKED ARRAY OF CHARFSTRING(n)	STRING(n)
SQL	PIC X CHAR	CHAR(n) PIC X(n)	VARCHAR(n)
TAL and pTAL	STRING	STRING array	--
Return Value Size (Words)	1	1 or 2, depends on declared pointer size	1 or 2, depends on declared pointer size

[1] HP COBOL85 only.

Table 78: Structured, Logical, Set, and File Types

	Byte-Addressed Structure	Word-Addressed Structure	Logical (true or false)	Boolean	Set	File
C-series BASIC	--	MAP buffer	--	--	--	--
C and C++	--	struct	--	--	--	--
COBOL	--	01-level RECORD	--	--	--	--
FORTRAN	RECORD	--	LOGICAL[1]	--	--	--
D-series Pascal	RECORD, byte-aligned	RECORD, word-aligned	--	BOOLEAN	Set	File
SQL	--	--	--	--	--	--

Table Continued

	Byte-Addressed Structure	Word-Addressed Structure	Logical (true or false)	Boolean	Set	File
TAL and pTAL	Byte-addressed standard STRUCT pointer	Word-addressed standard STRUCT pointer	--	--	--	--
Return Value Size (Words)	1 or 2, depends on declared pointer size	1 or 2, depends on declared pointer size	1 or 2, depends on compiler directive	1	1	1

[1] LOGICAL is normally defined as 2 bytes. The LOGICAL*2 and LOGICAL*4 compiler directives redefine LOGICAL.

Table 79: Pointer Types

	Procedure Pointer	Byte Pointer	Word Pointer	Extended Pointer
C-series BASIC	--	--	--	--
C and C++	function pointer	byte pointer	word pointer	extended pointer
COBOL	--	--	--	--
FORTRAN	--	--	--	--
D-series Pascal	Procedure pointer	Pointer, byte-addressed BYTEADDR	Pointer, byte-addressed WORDADDR	Pointer, extended-addressed EXTADDR
SQL	--	--	--	--
TAL	--	16-bit pointer, byte-addressed	16-bit pointer, word-addressed	32-bit pointer
pTAL	PROCPTR	16-bit pointer, byte-addressed	16-bit pointer, word-addressed	32-bit pointer
Return Value Size (Words)	1 or 2, depends on declared pointer size	1 or 2, depends on declared pointer size	1 or 2, depends on declared pointer size	1 or 2, depends on declared pointer size

Table 80: Address Types[1]

	Procedure Pointer	Byte Address	Word Address	Extended Address
C-series BASIC	--	--	--	--
C	--	--	--	--

Table Continued

	Procedure Pointer	Byte Address	Word Address	Extended Address
COBOL	--	--	--	--
FORTRAN	--	--	--	--
D-series Pascal	--	--	--	--
SQL	--	--	--	--
pTAL	PROCADDR	BADDR SGBADDR SGXBADDR CBADDR	WADDR SGWADDR SGXWADDR CWADDR	EXTADDR
Return Value Size (Words)	1 or 2, depends on declared pointer size	1 or 2, depends on declared pointer size	1 or 2, depends on declared pointer size	1 or 2, depends on declared pointer size

[1] Only the pTAL and EpTAL compilers support address types.

Features and Keywords of Version 2 Native C ++

Features Supported in VERSION2

These features are accepted in the **VERSION2** on page 324 dialect of native C++. These features are not in the *The Annotated C++ Reference Manual* (Ellis and Stroustrup) but are in the X3J16/WG21 Working Paper [The relevant sections of the WP are cited inside brackets]:

- The dependent statement of an `if`, `while`, `do-while`, or `for` is considered to be a scope, and the restriction on having such a dependent statement be a declaration is removed. [6.4 Selection statements]
- The expression tested in an if, while, do-while, or for, as the first operand of a "`?`" operator, or as an operand of the "`&&`", "`||`", or "`!`" operators can have a pointer-to-member type or a class type that can be converted to a pointer-to-member type in addition to the scalar cases permitted by the ARM. [6.4 Selection statements; 4.12 Boolean conversions; 6.4.1 The if statement; 6.4.2 The `switch` statement; 6.5 The iteration statement; 5.14 Logical `and` operator; 5.15 Logical `or` operator; 5.16 Conditional operator; 5.3.1 (7) Unary operators]
- Qualified names are allowed in elaborated type specifiers. [3.4.4 Elaborated type specifier]
- A global-scope qualifier is allowed in member references of the form `x.::A::B` and `p->::A::B`. [5.1 Primary expressions; 5.2.5 Class member access]
- The precedence of the third operand of the "`?`" operator is changed. [5.16 Conditional operator]
- If control reaches the end of the `main()` routine, and `main()` has an integral return type, it is treated as if a return 0; statement were executed. [3.6.1 (5) Main function]
- A functional-notation cast of the form `A()` can be used even if `A` is a class without a (nontrivial) constructor. The temporary object created gets the same default initialization to zero as a static object of the class type. [5.2.3 Explicit type conversion (function notation)]
- A cast can be used to select one out of a set of overloaded functions when taking the address of a function. [13.4 Address of overloaded functions]
- Type template parameters are permitted to have default arguments. [14.1 Template parameters]
- Function templates may have nontype template parameters. [14.1 Template parameters]
- A reference to `const volatile` cannot be bound to an `rvalue`. [8.5.3 References]
- Qualification conversions such as conversion from `T**` to `T const * const *` are allowed. [4.4 Qualification conversion]
- Digraphs are recognized. [2.5 Alternative tokens]
- Operator keywords (`and`, `bitand`, and so on) are recognized. [2.5 Alternative tokens]
- Static data member declarations can be used to declare member constants. [9.4.2 (4) Static data members]
- `wchar_t` is recognized as a keyword and a distinct type. [3.9.1(7) Fundamental types]
- `bool` is recognized. [3.9.1(8) Fundamental types]

- RTTI (run-time type identification), including `dynamic_cast` and `typeid` operator, is implemented. [5.2.8 Type identification]
- Declarations in tested conditions (in `if`, `switch`, `for`, and `while` statements) are supported. [6.4(3) Selection statements]
- Array `new` and `delete` are implemented. [5.3.4 New; 5.3.5 Delete; 18.4.1.2 Array forms]
- New-style casts (`static_cast`, `reinterpret_cast`, and `const_cast`) are implemented. [5.2.6 Dynamic cast; 5.2.8 Static cast; 5.2.9 Reinterpret cast; 5.2.10 Const cast]
- Definition of a nested class outside its enclosing class is allowed. [9.7(3) Nested class declarations]
- `mutable` is accepted on nonstatic data member declarations. [7.1.1 (9) Storage class specifiers]
- Namespaces are implemented, including using declarations and directives. Access declarations are broadened to match the corresponding using declarations. [7.3 Namespaces]
- Explicit instantiation of templates is implemented. [14.7.2 Explicit instantiation]
- The `typename` keyword is recognized. [7.1.5.3 Elaborated type specifiers]
- `explicit` is accepted to declare nonconverting constructors. [7.1.2 Function specifiers; 12.3.1 Conversion by constructor]
- The scope of a variable declared in the `for-init`-statement of a `for` loop is the scope of the loop (not the surrounding scope). [6.5.3 (4) The for statement]
- The new specialization syntax (using "template <>") is implemented. [14.7 Template specialization]
- The distinction between trivial and nontrivial constructors has been implemented, as has the distinction between PODs (plain ol' data) and non-PODs with trivial constructors. [12.1 (4, 5) Constructors]
- The linkage specification is treated as part of the function type (affecting function overloading and implicit conversions). [7.5 Linkage specifiers]
- `extern` inline functions are supported, and the default linkage for inline functions is external. [7.1.2 (4) Function specifiers]
- A `typedef` name can be used in an explicit destructor call. [7.1.3 The typedef specifier, 9.1 Class names; 12.4 Destructors]
- Pointers to arrays with unknown bounds as parameter types are diagnosed as errors. [8.3.5 Functions]
- Template friend declarations and definitions are permitted in class definitions and class template definitions. [14.5.3 Friends]
- Member templates are implemented. [14.5.2 Member templates]
- `Cv`-qualifiers are retained on `rvalues` (in particular, on function return values). [3.10 Lvalues and rvalues]
- Placement delete is implemented. [18.4.1.3 Placement forms]
- An array allocated via a `placement new` can be deallocated via `delete`. [18.4.1.2 Array forms]
- Covariant return types on overriding virtual functions are supported. [10.3 (5) Virtual functions]
- `enum` types are considered to be nonintegral types. [7.2 Enumeration types]
- Partial specialization of class templates is implemented. [14.5.4 Class template partial specialization]

- Partial ordering of function templates is implemented. [14.5.5.2 Partial ordering of function templates]
- Function declarations that match a function template are regarded as independent functions, not as "guiding declarations" that are instances of the template. [14.5.5 Function templates]

Features Not Supported in VERSION2

These features are not accepted in the VERSION2 dialect of native C++. These features are not in the *The Annotated C++ Reference Manual* (Ellis and Stroustrup) but are in the X3J16/WG21 Working Paper (WP) [The relevant sections of the WP are cited inside brackets]:

- It is not possible to overload operators using functions that take enum types and no class types. [13.3.1.2 Operators in expressions]
- enum types cannot contain values larger than can be contained in an int. [7.2 (5) Enumeration types]
- reinterpret_cast does not allow casting a pointer to member of one class to a pointer to member of another class if the classes are unrelated. [5.2.10 Reinterpret cast]
- Explicit qualification of template functions is not implemented. [14.6 Name resolution]
- In a reference of the form f()->g(), with g a static member function, f() is not evaluated. This is as required by the ARM. The WP, however, requires that f() be evaluated. [9.4 Static members]
- Putting a try/catch around the initializers and body of a constructor is not implemented. function-try-block is not implemented. [15 Exception handling]
- The new lookup rules for member references of the form x.A::B and p->A::B are not yet implemented. [3.4.3 Qualified name look up]
- The notation :: template (and ->template, and so on) is not implemented [3.4.5 Class member access]
- Template template parameters are not implemented. [14.1 Template parameters]
- Finding friend functions of the argument class types on name lookup on the function name in calls is not implemented. [14.5.3 Friends]

Keywords Added for the D45 Product Release

These keywords were introduced at the D45 release. Existing TNS/R native programs that use any of these keywords as identifiers must be changed to rename these identifiers.

```
and
and_eq
bitand
bitor
bool
catch
compl
dynamic_cast
explicit
false
mutable
namespace
not
not_eq
or
```

```
or_eq
throw
true
try
typeid
typename
using
wchar_t
xor
xor_eq
```

Defining Virtual Function Tables

Virtual function tables are defined when virtual functions are defined. (These tables are merely declared when there are virtual function declarations.) If a program declares but does not define a virtual function, the declaration exists without the required definition of its corresponding virtual function table (as per the standard).

The native C++ compiler generates a virtual function table for only one translation unit in a program when at least one virtual function is not defined with its class definition. The translation unit selected is the one that contains the definition of the lexically first non-inline virtual member function. Duplicate virtual function tables are generated for classes that define all virtual member functions within the class definition.

Example

```
class A {
   virtual int foo ();  // foo not defined within its class
 }; // A's virtual function table will be defined within the
    // module that contains the definition of A::foo.

class B{
   virtual int foo() {return 1;}
}; // all of B's virtual functions are defined within
   // B's definition, so each module that contains B also
   // contains a definition for B's virtual function table.
```

MIGRATION_CHECK Messages

MIGRATION_CHECK Warning Messages lists the warning messages emitted by the native C++ compiler when the pragmas `VERSION2` and `MIGRATION_CHECK` are specified in the RUN command for the compiler. Messages listed here are current at time of publication; see the appropriate `VERSION2` headers online for current messages on your system.

Using `MIGRATION_CHECK` causes the compiler to examine the source file to find instances of functions from the `VERSION2` C++ library that are not supported in the `VERSION3` library. The warning messages highlight elements that need to be examined and rewritten using the `VERSION3` library. The compiler does not generate an object file when `MIGRATION_CHECK` is specified.

Code Examples on page 597show a “Hello World” C++ program written in `VERSION2` code and in `VERSION3` code, in addition to the `MIGRATION_CHECK` output for the `VERSION2` code.

For more details, see the:

- pragma **MIGRATION_CHECK** on page 268
- pragma **VERSION3** on page 326

Table 81: MIGRATION_CHECK Warning Messages

No.	Class/Member Function Name	Warnings Displayed	Notes
algorithm: VERSION3 header file			
User programs should not make direct calls to any of the functions whose names have a prefix of a double-underscore (__) or a single-underscore (_). These are helper functions for internal use only. A warning is issued if any of these functions are used.			
bitset: VERSION3 header file			
1	class bitset	1. ‘bitset (const bitset<N>& rhs) throw((out_of_range, invalid_argument))' is not supported in version 3.	
		2. bool valid_position (size_t pos) const throw()' is not supported in version 3.	
		3. 'unsigned long index (size_t pos) const throw()' is not supported in version 3.	

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
		4. 'const size_t bitset_size' is is not supported in version 3.	
		5. 'bool valid_position(size_t)' is not supported in version 3 library.	
		6. 'unsigned long index(size_t)' is not supported in version 3 library.	
		7. Field 'size_t bitset::bistset_size' is not supported in version 3 library.	
		8. Constructor 'bitset::bitset(const bitset<N>&)' is not supported in version 3 library.	
		9. Operator& is defined as 'bitset<N> bitset::operator&(const bitset<N>)' in version 3 library.	
		10. Operator| is defined as 'bitset<N> bitset::operator|(const bitset<N>)' in version 3 library.	
		11. Operator^ is defined as 'bitset<N> bitset::operator^(const bitset<N>) in version 3 library.	
2	class reference (inner class)	None	

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
3	Template<size_t N> inline bitset <N> operator& (const bitset<N>& lhs, const bitset<N>& rhs) RWSTD_THROW_ SPEC_NULL	1. 'Operator & is defined as 'bitset<N> bitset::operator&(const bitset<N>)' in Version 3.	1. Equivalent general function is not defined in Version 3.
4	Template<size_t N> inline bitset <N> operator| (const bitset<N>& lhs, const bitset<N>& rhs) RWSTD_THROW_SPEC_NULL	1. 'Operator | is defined as 'bitset<N> bitset::operator|(const bitset<N>)' in Version 3.	1. Equivalent general function is not defined in Version 3.
5	Template<size_t N> inline bitset <N> operator^ (const bitset<N>& lhs, const bitset <N>& rhs) RWSTD_THROW_SPEC_NULL	1. 'Operator ^ is defined as 'bitset<N> bitset::operator^(const bitset<N>)' in Version 3.	1. Equivalent general function is not defined in Version 3.
common.h: VERSION3 header file			
1	struct common	1. "struct common not supported in Version 3.'	Equivalent file common.h is not defined in Version 3.
complex: VERSION3 header file			

	class complex<T>
	class complex<float>
	class complex<double>
	class complex<long double>
		None	All classes are supported in Version 3.

deque: VERSION3 header file			
1	class deque	1. 'allocator_type the_allocator' is not supported in Version 3.	1. In Version 3, Alloc Al can be used instead.
		2. 'iterator start' is not supported in version 3.	2. No equivalent fields defined in Version 3.
		3. 'iterator finish' is not supported in version 3.	3. No equivalent fields defined in Version 3.

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
		4. 'size_type length' is not supported in version 3.	4. In Version 3, the equivalent is size_type _Mysize.
		5. 'map_pointer map' is not supported in version 3.	5. Version 3 type and variables are: Mapptr Map.
		6. 'size_type map_size' is not supported in version 3.	6. size_type _Mapsize in Version 3.
		7. 'void allocate_at_begin()' is not supported in version 3.	7. Equivalents not defined.
		8. 'void allocate_at_end ()' is not supported in version 3.	8. Equivalent not defined in Version 3.
		9. 'void deallocate_at_begin()' is not supported in version 3.	9. Equivalent not defined in Version 3.
		10. 'void deallocate_at_end()' is not supported in version 3.	10. Equivalent not defined in Version 3.
		11. 'void insert_aux (iterator position, size_type n, const T& x)' is not supported in version 3.	11. Equivalent not defined in Version 3.
		12. 'void init_aux (InputIterator first, InputIterator last, _RW_is_not_integer)' is not supported in version 3.	12. No equivalent function defined in Version 3.
		13. 'void init_aux (InputIterator first, InputIterator last, _RW_is_integer)' is not supported in version 3.	13. No equivalent function defined in version 3.

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
		14. 'void insert_aux (iterator position, InputIterator first, InputIterator last, _RW_is_not_integer)' is not supported in version 3.	14. No equivalent function defined in Version 3.
		15. 'void insert_aux (iterator position, InputIterator first, InputIterator last, _RW_is_integer)' is not supported in version 3.	15. No equivalent function defined in Version 3.
		16. ' void insert_aux2 (iterator position, InputIterator first, InputIterator last)' is not supported in version 3.	16. No equivalent function defined in Version 3.
		17. 'void insert_aux2 (iterator position, const_iterator first, const_iterator last)' is not supported in version 3.	17. No equivalent function defined in Version 3.
		18. 'void insert_aux2 (iterator position, const T* first, const T* last)' is not supported in version 3.	18. No equivalent function defined in Version 3.
		19. 'deque (const T* first, const T* last)' is not supported in version 3.	19. No equivalent constructor is defined in Version 3.
		20. 'deque (const T* first, const T* last, const Allocator& alloc RWSTD_DEFAULT_ARG(Allocator()))' is not supported in Version 3.	20. No equivalent constructor is defined in Version 3.
2	Class const_iterator (inner class)	21. 'const_iterator (const iterator& x)' is not supported in version 3.	21. No equivalent constructor in Version 3.

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
		22. 'const_iterator (pointer x, map_pointer y)' is not supported in version 3.	22. No equivalent constructor is defined in Version 3.
		23. 'pointer current' is not supported in version 3.	23. No equivalent field is defined in Version 3.
		24. 'pointer first' is not supported in version 3.	24. No equivalent field is defined in Version 3.
		25. 'pointer last' is not supported in version 3.	25. No equivalent field is defined in Version 3.
		26. 'map_pointer node' is not supported in version 3.	26. No equivalent field is defined in Version 3.
		27. 'iterator (pointer x, map_pointer y)' is not supported in version 3.	27. In Version3, class iterator is derived from class const_iterator.
		28. 'pointer current' is not supported in version 3.	28. No equivalent constructor in Version 3.
		29. 'pointer first' is not supported in version 3.	29. No equivalent field is defined in Version 3.
	Class iterator (inner class)	30. 'pointer last' is not supported in version 3.	30. No equivalent field is defined in Version 3.
		31. 'map_pointer node' is not supported in version 3.	31. No equivalent field is defined in Version 3.
exception: VERSION3 header file			
1	Class bad_exception	None	This class is supported in Version 3.
fstream/ fstream.h: VERSION3 header file			
1	filebuf	1. 'filebuf(int fd, char* p, int l)' is not supported in version 3.	1. 'fEquivalent class in Version 3 is 'basic_filebuf' type defined to 'filebuf'.

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
2	fstreambase	1. 'class fstreambase' is not supported in version3 library	1. 'fNo equivalent class is defined in Version 3.
3	Ifstream	1. 'ifstream(int fd, char* p, int l)' is not supported in version 3.2. 'class fstreambase' is not supported in version3.	1. Equivalent class in version 3 is 'basic_ifstream' type defined to 'ifstream'.2. No equivalent constructor is Version 3.
4	ofstream	1. 'ofstream(int fd, char* p, int l)' is not supported in version 3.	1. Equivalent class in Version 3 is 'basic_ofstream' type defined to 'ofstream'.
		2. Class 'fstreambase' is not supported in version3 library.	2. No equivalent constructor is defined in Version 3.
5	fstream	1. 'ifstream(int fd, char* p, int l)' not supported in version 3.	1. Equivalent class in Version 3 is 'basic_fstream' type defined to 'fstream'.
		2. Class 'fstreambase' is not supported in version3 library.	2. No equivalent constructor in Version 3.
functional: VERSION3 header file			
1	struct times	1. 'struct times' is not supported in version 3.	1. No equivalent structure is defined in Version 3.
1	template<class Arg, class Result>struct unary_function		No warning issued.
2	template<class Arg1, class Arg2, class Result>struct binary_function		No warning issued.
3	template<class T>struct plus		No warning issued.
4	template<class T>struct minus		No warning issued.
5	template<class T>struct multiplies		No warning issued.
6	template<class T>struct divides		No warning issued.
7	template<class T>struct modulus		No warning issued.

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
8	template<class T>struct negate		No warning issued.
9	template<class T>struct equal_to		No warning issued.
10	template<class T>struct not_equal_to		No warning issued.
11	template<class T>struct greater		No warning issued.
12	template<class T>struct less		No warning issued.
13	template<class T>struct greater_equal		No warning issued.
14	template<class T>struct less_equal		No warning issued.
15	template<class T>struct logical_not		No warning issued.
16	template <class Operation>class binder1st		No warning issued.
17	template <class Operation>class binder2nd		No warning issued.
18	template<class Operation1, class Operation2>class unary_compose		No warning issued.
19	template <class Arg, class Result>class pointer_to_unary_function		No warning issued.
20	template <class Arg, class Result>class pointer_to_binary_function		No warning issued.
iomanip/iomanip.h: VERSION3 header file			
1	class SMANIP	None.	No equivalent class is defined in Version 3.
2	class SAPP	1. 'class SAPP' is not supported in Version 3.	No equivalent class is defined in Version 3.
3	class IMANIP	2. 'class IMANIP' is not supported in Version 3.	No equivalent class is defined in Version 3.
4	class IAPP	3. 'class IAPP' is not supported in Version 3.	No equivalent class is defined in Version 3.
5	class OMANIP	4. 'class OMANIP' is not supported in Version 3.	No equivalent class is defined in Version 3.
6	class OAPP	5. 'class OAPP' is not supported in Version 3.	No equivalent class is defined in Version 3.

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
7	class IOMANIP	6. 'class IOMANIP' is not supported in Version 3.	No equivalent class is defined in Version 3.
8	class IOAPP	7. 'class IOAPP' is not supported in Version 3.	No equivalent class is defined in Version 3.
iostream/iostream.h: VERSION3 header file			
1	class ios	1. 'class ios' is defined inside namespace std in version 3.	All template classes of Version 3 are defined within a namespace `std`. In Version 2 no namespace is defined.
			1. Equivalent class 'basic_ios' is defined in file - ios type defined to 'ios'.
2	class streambuf	2. 'class streambuf' is defined inside namespace std in version 3.	2. Equivalent class 'basic_streambuf' is defined in file -streambuf type defined to streambuf.
		3. 'virtual int doallocate()' is not supported in version 3.	3. No equivalent function in Version 3.
		4. 'int optim_in_avail()' is not supported in version 3.	4. No equivalent function in Version 3.
		5. 'int optim_sbumpc()' is not supported in version 3.	5. No equivalent function in Version 3.
		6. 'int unbuffered() const' is not supported in version 3.	6. No equivalent function in Version 3.
		7. 'void unbuffered(int unb)' is not supported in version 3.	7. No equivalent function in Version 3.
		8. 'void dbp()' is not supported in version 3.	8. No equivalent function in Version 3.

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
		9. 'streambuf* setbuf(char* p, int len, int count)' is not supported in version 3.	9. No equivalent function is defined in Version 3.
3	class istream	10. 'class istream' is defined inside namespace std in version 3.	10. Equivalent class 'basic_istream' is defined in file -istream type defined to 'istream'.
		11. 'istream& rs_complicated(unsigned char& c)' is not supported in version 3.	11. No equivalent function is defined in version 3.
		12. 'istream& rs_complicated(char& c)' is not supported in version 3.	12. No equivalent function is defined in Version 3.
		13. ' istream& get_complicated(unsigned char& c)' is not supported in version 3.	13. No equivalent function is defined in Version 3.
		14. ' istream& get_complicated(char& c)' is not supported in version 3.	14. No equivalent function is defined in Version 3.
4	class ostream	15. 'class ostream' is defined inside namespace std in version 3.	15. Equivalent class 'basic_ostream' is defined in file -ostream type defined to 'ostream'.
		16. 'ostream& complicated_put(char c)' is not supported in Version 3.	16. No equivalent function is defined in Version 3.
		17. 'ostream& ls_complicated(char)' not supported in Version 3.	17. No equivalent function is defined in class reference (inner class Version 3.
		18. 'ostream& ls_complicated(unsigned char)' is not supported in Version 3.	18. No equivalent function is defined in Version 3.

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
5	class iostream	19. 'class iostream' is defined inside namespace std in version 3.	19. Equivalent class 'basic_iostream' is defined in file - istream type defined to 'iostream'.
6	class istream_ withassign	20. class istream_withassign' is not supported in version 3 library.	20. No equivalent class is defined in Version 3.
7	class osrteam_ withassign	21. class 'ostream_withassign' is not supported in version 3 library.	21. No equivalent class is defined in Version 3.
8	class iostream_ withassign	22. class 'iostream_withassign' is not supported in version 3 library.	22. No equivalent class is defined in Version 3.
9	static class iostream_init	23. class 'iostream_init' is not supported in version 3 library.	23. No equivalent class is defined in Version 3.
iterator: VERSION3 header file			
1	Struct input_iterator	1. 'struct input_iterator' is not supported in version 3.	1. No equivalent structure is defined in Version 3.
2	struct output_iterator	2. 'struct output_iterator' is not supported in version 3.	2. Equivalent structure '_Outit' is defined in xutility.
3	Struct forward_iterator	3. 'struct forwardt_iterator' is not supported in version 3.	3. No equivalent structure is defined in Version 3.
4	Struct bi-directional_ iterator	4. 'struct bidirectional_iterator' is not supported in version 3.	4. No equivalent structure is defined in Version 3.
5	Struct random_access_ iterator	5. 'struct random_access_iterator' is not supported in version 3.	5. No equivalent structure is defined in Version 3.

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
6	class reverse_bi-directional_ iterator	6. 'class bidirectional_iterator<class T,class Distance>' is not supported in version 3.	6. Extends struct bidirectional_iterator in Version2.
7	class reverse_iterator	7. 'struct random_access_iterator <class T,class Distance>' is not supported in version 3.	7. Extends struct random_access_ iterator in Version 2.
8	class back_insert_ iterator	8. 'struct output_iterator' is not supported in version 3.	8. Extends struct output_iterator in Version 2
9	class front_insert_ iterator	9. 'struct output_iterator' is not supported in version 3.	9. Extends struct output_iterator in Version 2.
10	class insert_iterator	10. 'struct output_iterator' is not supported in version 3.	10. Extends struct output_iterator in Version 2.
11	class istream_iterator	11. 'struct input_iterator<T,Distance>' is not supported in version 3.	11. Extends struct input_iterator in Version 2.
12	class ostream_iterator	12. struct output_iterator' is not supported in version 3.	12. Extends struct output_iterator in Version 2.
13	template <class InputIterator, class Distance> inline void distance (InputIterator first, InputIterator last, Distance& n)	1. 'void distance (InputIterator first, InputIterator last, Distance& n)' is not supported in Version 3.	1. No equivalent function is defined in Version 3.
14	template <class InputIterator, class Distance> inline void advance (InputIterator& i, Distance n)	1. 'void advance (Inputterator& i, Distance n)' is not supported in Version 3.	1. No equivalent function is defined in Version 3.
limits: VERSION3 header file			
Supported templates and classes:			Equivalent implementation of

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
1	template <class T> class RWSTDHuge numeric_limits		<limits2> header in version 3 is <limits>. No warning messages are needed or added for the classes and methods in this file.
2	template <class T> class RWSTDHuge numeric_limits		
3	class RWSTDExport numeric_limits <float>		
4	class RWSTDExport numeric_limits <double>		
5	class RWSTDExport numeric_limits <long double>		
6	class RWSTDExport numeric_limits <int>		
7	class RWSTDExport numeric_limits <wchar_t>		
8	class RWSTDExport numeric_limits <short>		
9	class RWSTDExport numeric_limits <unsigned short>		
10	class RWSTDExport numeric_limits <unsigned int>		
11	class RWSTDExport numeric_limits <long>		
12	class RWSTDExport numeric_limits <unsigned long>		
13	class RWSTDExport numeric_limits <unsigned char>		
14	class RWSTDExport numeric_limits <signed char>		
15	class RWSTDExport numeric_limits <bool>		
list: VERSION3 header file			
1	class list	1. 'void assign (Size n)' is not supported in version 3.	1. 'No equivalent function in Version 3.

	Equivalent implementation of file
	`list2`
	in Version 2 is
	`list`
	in Version 3.
		2. 'void assign (size_type n)' is not supported in version 3.	2. 'No equivalent function in Version 3.

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
		3. 'size_type allocation_size()' is not supported in version 3.	3. 'No equivalent function in Version 3.
		4. 'size_type allocation_size(size_type new_size)' is not supported in version 3.	4. 'No equivalent function in Version 3.
		5. 'void add_new_buffer (size_type n)' is not supported in version 3.	5. 'No equivalent function in Version 3.
		6. 'void deallocate_buffers ()' is not supported in version 3.	6. 'No equivalent function in Version 3.
		7. 'link_type get_node (size_type n)' is not supported in version 3.	7. 'No equivalent function in Version 3.
		8. 'void put_node (link_type p)' is not supported in version 3.	8. 'No equivalent function in Version 3.
		9. 'void init(size_type n)' is not supported in version 3.	9. 'No equivalent function in Version 3.
		10. 'void init_aux (InputIterator first, InputIterator last, _RW_is_not_integer)' is not supported in version 3.	10. 'No equivalent function in Version 3.
		11. 'void init_aux (InputIterator first, InputIterator last, _RW_is_integer)' is not supported in version 3.	11. 'No equivalent function in Version 3.
		12. 'void insert_aux (iterator position, size_type n, const T& x)' is not supported in version 3.	12. 'No equivalent function in Version 3.

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
		13. 'void insert_aux (iterator position, InputIterator first, InputIterator last, _RW_is_not_integer)' is not supported in version 3.	13. 'No equivalent function in Version 3.
		14. 'void insert_aux (iterator position, InputIterator first, InputIterator last, _RW_is_integer)' is not supported in version 3.	14. 'No equivalent function in Version 3.
		15. 'void insert_aux2 (iterator position, InputIterator first, InputIterator last)' is not supported in version 3.	15. 'No equivalent function inVersion 3.
		16. 'void insert_aux2 (iterator position, const_iterator first, const_iterator last)' is not supported in version 3.	16. 'No equivalent function in Version 3.
		17. 'void transfer (iterator position, iterator first_, is not supported in version 3.	17. 'No equivalent function in Version 3.
		18. 'void set_allocator(allocator_type a)' is not supported in version 3.	18. 'No equivalent function in Version 3.
		19. 'size_type buffer_size' is not supported in version 3.	19. 'No equivalent function in Version 3.
		20. 'Allocator the_allocator' is not supported in version 3.	20. 'No equivalent function in Version 3.
		21. 'buffer_pointer buffer_list' is not supported in version 3.	21. 'No equivalent function in Version 3.

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
		22. 'link_type free_list' is not supported in version 3.	22. 'No equivalent function in Version 3.
		23. 'link_type next_avail' is not supported in version 3.	23. 'No equivalent function in Version 3.
		24. 'link_type last' is not supported in version 3.	24. 'No equivalent function in Version 3.
		25. 'link_type node' is not supported in version 3.	25. 'No equivalent function in Version 3.
		26. 'size_type length' is not supported in version 3.	26. 'No equivalent function in Version 3.
2	struct list_node (inner Structure)	27. struct list_node not supported in version 3.	27. Equivalent 'struct _Node' is defined in version 3.
3	struct list_node_buffer (inner structure)	28. struct list_node_buffer not supported in version 3.	28. Equivalent 'struct _Node' is defined in Version 3.
4	class iterator (inner class)	None.	None.
5	class const_iterator (inner class)	None.	None.
map: VERSION3 header file			
1	struct select1st	1. 'struct select1st' is not supported in version 3.	1. No equivalent structure is defined in Version 3.
2	class map	2. 'map (const map<Key, T, Compare, Allocator>& x) ' is not supported in version 3.	1. No equivalent constructor is defined in version 3.
		3. ' size_type allocation_size ()' is not supported in version 3.	2. No equivalent finction in Version 3.

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
		4. ' size_type allocation_size(size_type new_size)' is not supported in version 3.	3. No equivalent finction is defined in Version 3.
3	class value_compare (inner class)	5. 'Compare RWSTD_COMP' not supported in version 3.	5. No equivalent field is defined in Version 3.
4	class multimap	6. 'multimap (const multimap<Key, T, Compare, Allocator>& x)' is not supported in version 3.	6. No equivalent constructor is defined in Version 3.
		7. 'size_type allocation_size()' is not supported in version 3.	7. No equivalent finction in Version 3.
		8. 'size_type allocation_size(size_type new_size)' is not supported in version 3.	8. No equivalent finction is defined in Version 3.
		9. 'Compare RWSTD_COMP' not supported in version 3.	9. No equivalent field is defined in Version 3.
memory and xmemory: VERSION3 header files			
1	struct__FS	1. 'struct __FS' is not supported in version 3.	1. No equivalent structis defined in Version 3.
2	Class RWSTD-Export bad_alloc	None.	None.
3	class raw_storage_ iterator	3. 'OutputIterator iter' is not supported in version 3.	3. No equivalent field is defined in Version 3.
4	class RWSTDExport allocator	None.	None.
5	class allocator_interface	1. 'class allocator_interface' is not supported in version 3 library.	5. Equivalent classes are defined in Version 3.

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
		2. 'allocator_type alloc_' is not supported in Version 3.	
		3. 'inline void allocator_interface<Allocator,T>:: construct(pointer p, const T& val)' is not supported in Version 3.	
		4. 'inline void allocator_interface<Allocator,T>:: destroy(T* p)' is not supported in Version 3.	
6	class allocator_interface<allocator,void>	6. 'class allocator_interface<allocator, void>' is not supported in Version 3.	6. No equivalent class is defined in Version 3.
7	class auto_ptr	7. 'explicit auto_ptr (X* p = 0)' is not supported in version 3.	7. No equivalent function in Version 3.
		8. 'auto_ptr (const auto_ptr<X>& a)' is not supported in version 3.	8. No equivalent function in Version 3.
		9. 'auto_ptr<X>& operator= (const auto_ptr<X>& rhs)' is not supported in version 3.	9. No equivalent function in Version 3.
		10. 'X& operator* () const' is not supported in version 3.	10. No equivalent function in version 3.
		11. 'X* operator-> () const' is not supported in version 3.	11. No equivalent function in version 3.
		12. 'X* get () const' is not supported in version 3.	12. No equivalent function in Version 3.
		13. 'X* release ()' is not supported in version 3.	13. No equivalent function in Version 3.

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
8	1. template <class ForwardIterator> RWSTD_TRICKY_INLINE void destroy (ForwardIterator first, ForwardIterator last)	1. 'void destroy (ForwardIterator first, ForwardIterator last)' is not supported in Version 3.	1. No equivalent function in Version 3.
new: VERSION3 header file			
1	class bad_alloc	None.	Equivalent implementation of <new> header is in Version 3 <new>.
numeric: VERSION3 header file			
Equivalent implementation of <numeric2> header is in version 3 <numeric>. All functions are supported in Version 3.			
queue: VERSION3 header file			
1	class queue	1. Class queue : Template Mismatch	Version 2 library: template <class T, class Container, class Allocator>.
			Version 3 library: template<class _Ty, class _Container = deque<_Ty>.
2	class priority_queue	2. 'class priority_queue : Template Mismatch'	Template in Version 2 library supports at most four parameters:
			template<class T, class Container = vector<T>, class Compare = less<typename
			Container::value_ type>, class Allocator = allocator >.Template in Version 3 library supports at most three parameters:

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
			template<class _Ty, class _Container = vector<_Ty>, class _Pr = less <typename _Container::value_ type>.
3	template <class T, class Container, class Allocator> inline bool operator== (const queue<T,Container,Allocator>& x, const queue<T,Container,Allocator>& y)	1. Function operator == : Template mismatch	Template in Version 2 library supports at most four parameters: template< class T, class Container = vector<T>, class Compare = less<typename Container::value_type>, class Allocator = allocator >.
			Template in Version 3 library supports at most three parameters: template< class _Ty, class _Container = vector <_Ty>, class _Pr = less<typename _Container::value_type> >.
4	template <class T, class Container, class Allocator> inline bool operator< (const queue<T,Container,Allocator>& x, const queue<T,Container,Allocator>& y)	1. Function operator < : Template mismatch	Template in Version 2 library supports at most four parameters: template< class T, class Container = vector<T>, class Compare = less<typename Container::value_type>, class Allocator = allocator >.
			Template in Version 3 library supports at most three parameters: template< class _Ty, class _Container = vector<_Ty>,class _Pr = less<typename _Container::value_type> >.
rwexcept: VERSION2 header file			

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
		None.	Equivalent implementation of rwexcept is found in exception in Version 3.
rwnew: VERSION2 header file			
1	class RWSTDExport bad_alloc	None.	Equivalent implementation of rwnew is found in new in Version 3.
rwstdex: VERSION2 header file			
	1. class RWSTDExport logic_error		Version 3 implements all these classes in file stdexcept. No warning messages issued.
	2. class RWSTDExport domain_error		
	3. class RWSTDExport length_error		
	4. class RWSTDExport out_of_range		
	5. class RWSTDExport runtime_error		
	6. class RWSTDExport range_error		
	7. class RWSTDExport overflow_error		
set: VERSION3 header file			
1	struct ident	1. 'struct ident' is not supported in version 3.	1. No equivalent function in version 3.
2	class set	2. 'set (const set<Key, Compare, Allocator>& x)' is not supported in version 3.	2. No equivalent constructor is defined in Version 3.
		3. 'size_type allocation_size ()' is not supported in version 3.	3. No equivalent function in Version 3.
		4. 'size_type allocation_size(size_type new_size)' is not supported in version 3.	4. No equivalent class is defined in Version 3.
3	class multiset	5. 'multiset (const multiset<Key, Compare, Allocator>& x)' is not supported in version 3.	5. No equivalent constructor is defined in Version 3.

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
		6. 'size_type allocation_size()' is not supported in version 3.	6. No equivalent function in Version 3.
		7. 'size_type allocation_size(size_type new_size)' is not supported in version 3.	7. No equivalent function is defined in Version 3.
stack: VERSION3 header file			
1	class stack	1. class stack: Template Mismatch	Version 2 library: template <class T, class Container, class Allocator>.
			Version 3 library: template<class _Ty, class _Container = deque<_Ty>.
stdexcept2: VERSION2 header file			
1	class exception	None.	Equivalent implementation of stdexcept2 is found in exception in Version 3.
stdiostream: VERSION3 header file			
1	class stdiobuf	1. 'class stdiobuf' is defined inside namespace std in version 3.	Class stdiobuf extends streambuf. class 'basic_filebuf' is typedefined as stdiobuf in Version 3.
			In Version 3, class 'basic_filebuf' is defined in namespace std. In Version 2 no namespace is defined.
2	class stdiostream	2. 'class stdiostream' is defined inside namespace std in version 3.	Class stdiostream extends class ios. class name is 'basic_fstream' is typedefined to Version 3.

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
			In Version 3 class 'basic_fstream' is defined in namespace std. In Version 2 no namespace is defined.
string: VERSION3 header file			
1	template <class charT> struct RWSTDHuge string_char_traits	1. 'struct RWSTDHuge string_char_traits' is not supported in version 3.	
2	template <class charT> struct RWSTDExport string_char_traits<char>	2. 'struct RWSTDExport string_char_traits<char>' is not supported in version 3.	
3	template <class charT> struct RWSTDExport string_char_traits <wchar_t>	3. 'struct RWSTDExport string_char_traits<wchar _t>' is not supported in version 3.	
4	template <class charT> struct RWSTDExport-Template rw_traits	4. 'struct RWSTDExportTemplate rw_traits' is not supported in version 3.	
5	struct RWSTDExport rw_traits<char, string_char_traits<char> >	5. 'struct RWSTDExport rw_traits<char, string_char_traits<char> >' is not supported in version 3.	
6	struct RWSTDExport rw_traits<wchar_t, string_char_traits<wchar _t>>	6. 'struct RWSTDExport rw_traits<wchar_t, string_char_traits<wchar _t>>' is not supported in version 3.	
7	class RWSTDHuge string_ref_rep	7. 'class RWSTDHuge string_ref_rep' is not supported in version 3.	
8	struct RWSTDHuge null_string_ref_ rep	8. 'struct RWSTDHuge null_string_ref_rep' is not supported in version 3.	

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
9	class RWSTDHuge string_ref	9. 'class RWSTDHuge string_ref' is not supported in version 3.	
10	class RWSTDHuge basic_string	10. size_type getCapac() const' is not supported in version 3.	
		11. 'void clobber (size_type)' is not supported in version 3.	
		12. 'void cow()' is not supported in version 3.	
		13. 'void cow(size_type nc)' is not supported in version 3.	
		14. 'void init_aux(InputIterator first, InputIterator last, _RW_is_not_integer)' is not supported in version 3.	
		15. 'void init_aux(InputIterator first, InputIterator last, _RW_is_integer)' is not supported in version 3.	
strstream.h: VERSION3 header file			
1	class strstreambuf	1. 'class strstreambuf' is defined in namespace std in version 3.	1. Equivalent 'class strstreambuf' is defined in - strstream buf in Version 3.
		2. 'virtual int doallocate()' is not supported in version 3.	2. No equivalent function is defined in Version 3.
		3. 'int isfrozen()' is not supported in version 3.	3. No equivalent function is defined in Version 3.
			In Version 3 class 'strstreambuf' is defined in namespace std.

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
			In Version 2 no namespace is defined.
2	class strstreambase	4. 'class strstreambase' is not supported in version 3 library.	4. No equivalent class is defined in Version 3.
3	class istrstream	5. 'class istrstream' is defined in namespace std in version 3.	5. Equivalent 'class istrstream' is defined in strstream in Version 3.
		6. 'super class strstreambase' is not supported in version 3 library.	6. Not supported in Version 3.
4	class ostrstream	7. 'class ostrstream' is defined in namespace std in version 3.	7. Equivalent 'class ostrstream' is defined in strstream in Version 3.
		8. 'super class strstreambase' is not supported in version 3 library.	8. Not supported in Version 3.
5	class strstream string_char_traits<char>	9. 'class strstream' is defined in namespace std in version 3.	9. Equivalent 'class strstream' is defined in strstream in Version 3.
		10. 'super class strstreambase' is not supported in version 3 library.	10. Not supported in Version 3.
tree: VERSION3 header file			
1	class iteration (inner class)	None.	None.
2	class const_iterator (inner class)	None.	None.
typeinfo: VERSION3 header file			
1	class type_info	None.	Equivalent implementation of typeinfo2 can be found in file typeinfo in Version 3.
2	class bad_cast	None.	

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
3	class bad_typeid	None.	
utility: VERSION3 header file			
1	Constructs supported: template<classT1, class T2> struct pair	None.	Equivalent implementation of utility2 header is in file utility in Version 3.
vector: VERSION3 header file			
1	class vector	1. 'explicit vector (size_type n, const Allocator& alloc RWSTD_DEFAULT_ARG(Allocator()))' is not supported in version 3.	1. No equivalent constructor is defined in Version 3.
		2. 'buffer_size' is not supported in Version 3.	2. Equivalent variable is not supported in Version 3.
		3. 'allocator_type the _allocator' is not supported in Version 3.	3. Equivalent object is not supported in Version 3.
		4. 'iterator start' is not supported in Version 3.	4. Equivalent object is not supported in Version 3.
		5. 'iterator finish' is not supported in Version 3.	5. Equivalent object is not supported in Version 3.
		6. 'iterator end_of_storage' is not supported in Version 3.	6. Equivalent object is not supported in Version 3.
		7. 'void insert_aux (iterator position, InputIterator first, InputIterator last, _RW_ist_integer)' is not supported in version 3.	7. No equivalent function is defined in Version 3.
		8. 'void insert_aux2(iterator position, InputIterator first, InputIterator last)' is not supported in version 3.	8. No equivalent function is defined in Version 3.

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
		9. 'void insert_aux2(iterator position, const_iterator first, const_iterator last)' is not supported in version 3.	9. No equivalent function is defined in Version 3.
		10. 'void destroy(iterator start_, iterator finish_)' is not defined in version 3 library.	10. Equivalent function 'void _Destroy(pointer _First, pointer _Last)' is defined in Version 3.
		11. 'void assign (size n)' is not supported in version 3.	11. No equivalent function is defined in Version 3.
		12. 'void assign (size_type n)' is not supported in version 3.	12. No equivalent function is defined in Version 3.
		13. 'size_type allocation_size()' is not supported in version 3.	13. No equivalent function is defined in Version 3.
		14. 'size_type allocation_size(size_type new_size)' is not supported in version 3.	14. No equivalent function is defined in Version 3.
2	class RWSTDExport vector<bool, allocator>	1. 'iterator start' not supported in version 3.	1. No equivalent field is defined in Version 3.
		2. 'iterator finish' not supported in version 3.	2. No equivalent field in Version 3.
		3. 'unsigned int* end_of_storage' not supported in version 3.	3. No equivalent field is defined in Version 3.
3	class reference (inner class)	1. 'bool operator== (const reference& x) const' is not supported in version 3.	1. No equivalent function is defined in Version 3.
		2. 'bool operator< (const reference& x) const' is not supported in version 3.	2. No equivalent function is defined in Version 3.

Table Continued

No.	Class/Member Function Name	Warnings Displayed	Notes
		3. 'unsigned int* p' is not supported in Version 3.	3. Equivalent variable not supported.
		4. 'unsigned int mask' is not supported in Version 3.	4. Equivalent variable not supported.
4	class iterator (inner class)	1. 'iterator (unsigned int* x, unsigned int y)' is not supported in version 3.	1. No equivalent constructor in Version 3.
		2. 'void bump_up ()' is not supported in version 3.	2. No equivalent function in Version 3.
		3. 'void bump_down ()' is not supported in version 3.	3. No equivalent function in Version 3.
		4. 'unsigned int* p' is not supported in Version 3.	4. Equivalent variable not supported.
		5. 'unsigned int offset' is not supported in Version 3.	5. Equivalent variable not supported.
5	class const_iterator	1. 'const_iterator (unsigned int* x, unsigned int y)' is not supported in version 3.	1. No equivalent constructor in Version 3.
		2. 'void bump_up ()' is not supported in version 3.	2. No equivalent function in Version 3.
		1. 'void bump_down ()' is not supported in version 3.	3. No equivalent function in Version 3.
		4. 'unsigned int* p' is not supported in Version 3.	4. Equivalent variable not supported.
		5. 'unsigned int offset' is not supported in Version 3.	5. Equivalent variable not supported.

Code Examples

- For `VERSION3`, all classes defined in the C++ Standard are put into namespace `std`.
- For `VERSION2,`several classes were in global namespace.

`VERSION2` code for a simple “Hello world” program:

```
#include <iostream>
int main (void) {
     cout << “Hello world” << endl;
}
```

One way to use `VERSION3` for the “Hello world” program:

```
# include <iostream>
int main (void) {
     std::cout << “Hello world” << std::endl;
}
```

Alternative `VERSION3` code for the “Hello world” program:

```
#include <iostream>
using namespace std:
int main (void) {
     cout << “Hello world” << endl;
}
```

If you compile the `VERSION2` code as `VERSION3`, you will get two undefined identifiers (`cout` and `endl`). Similar errors occur if you compile the `VERSION3` source as `VERSION2`.

To compile the “Hello world” program (file name `test.cpp`) using the `MIGRATION_CHECK` pragma:

- On OSS environment:

  ```
  c89 test.cpp -Wversion2 -Wmigration_check -Wnosuppress
  ```

- On Guardian:

  ```
  nmcplus/ in testcpp, out $s.#list/ ; migration_check, version2
  ```

The listing from `MIGRATION_CHECK` will contain:

```
 5.   1  1   cout << “Hello world” << endl;
*** Warning: ^
--> (13) : ‘cout’ is in namespace std:: for version3
```

c99 Selected Features (C99LITE)

This appendix summarizes the subset of features from the 1999 update to the ISO/IEC standard for C (ISO/IEC 9899:1999, also known as c99) that are supported by the HPE C compiler on systems running H06.08 and later H-series RVUs and J06.03 and later J-series RVUs. By default, these features are not recognized by the C compiler; you must specify the C99LITE pragma on the compiler RUN command (Guardian) or the -Wc99lite option on the c89 command (OSS, Windows) to use these features. These features are supported by the TNS/E and TNS/X native C compilers; they are not supported by the TNS C, TNS/R C, or C++ compilers.

NOTE: H06.21 and later H-series RVUs and J06.10 and later J-series RVUs and L15.02 RVU also include full support for the c99 standard for TNS/E native applications that use IEEE floating point. Complex types and several new math functions and macros are not supported for Tandem floating point format, but all other c99 features are available when compiled using the Tandem floating point format option. For more information about this support, see **c99 Full Support** on page 602.

The relevant sections of the C standard are cited inside the brackets.

__func__ Identifier

The __func__ identifier provides a string representing the name of the current function, or an empty string if it appears outside of a function. An existing HPE extension, the __FUNCTION__ macro, provides the same functionality. [6.4.2.2, Predefined identifiers]

Example

```
#include <stdio.h>
void myfunc(void) {
  printf ("%s\n", __func__); // prints myfunc
}
```

Universal Character Names

A universal character name has one of these forms:

- \unnnn
- \Unnnnnnnn

where *n* is a hexadecimal digit.

Universal character names can be used in identifiers, character constants, and string literals to designate characters that are not in the basic character set, such as non-English characters. [6.4.3, Universal character names]

Example

```
char *ptr = "\u0024";
```

Hexadecimal Floating Point Constants

Floating point constants can be expressed using hexadecimal digits. [6.4.4.2, Floating constants]

Example

```
Float f = 0x3F.Cp1
```

Digraph Characters

Digraph characters (two-character sequences that represent a single character) can be used to represent the characters shown in the following table:

Character	Equivalent Digraph
[	<:
]	:>
{	<%
}	%>
#	%:
##	%:%:

The digraphs enable you to enter characters that are not allowed in certain rare circumstances. [6.4.6, Punctuators]

Example

```
iarr<:10:>
is equivalent to
iarr[10]
```

Comments

The characters // can be used to introduce a comment. The comment begins with the // and terminates at the end of the line. [6.4.9, Comments]

Example

```
/* This is a comment */
//And now so is this!
```

Implicit Function Declarations Not Allowed

Implicit function declaration is not allowed: every function must be explicitly declared before it can be called. Previously, implicit function declaration was allowed and caused the compiler to generate a warning message. Implicit function declaration now generates an error. [6.7.5.3, Function declarators]

Example

```
int main(void) {
   foo(); //error! not declared
   .
   .
```

```
    .
}
```

Designated Initialization for Structures

Aggregate initialization is enhanced to allow designators that initialize the components of an array or struct.

Example

```
typedef struct div_t {
   int quot;
   int rem;
} div_t;
div_t answer = {.quot = 2, .rem = -1};
```

The components of `div_t`, `quot` and `rem`, are initialized to 2 and -1, respectively.

Previously, you would initialize the struct components as follows:

```
typedef struct div_t {
   int quot;
   int rem;
} div_t;
div_t answer = {2, -1};
```

Nonconstant Aggregate Component Initializers

The initializer expression for an automatic composite variable can contain nonconstant subexpressions. [6.7.8, Initialization]

Example

```
typedef struct whole {
   int part1;
   int part2;
} whole;
void foo (int i, int j) {
   whole local_var = {i, j};
```

Previously, only constants were allowed in the initializer expressions; nonconstant initialization required an assignment statement.

Mixed Statements and Declarations

Declarations can appear within a block. Previously, this was an error. [6.8.2, Compound statement]

Example

```
{
   foo();
   int i = 1; //no longer an error
   bar(i);
}
```

New Block Scopes for Selection and Iteration Statements

All selection statements (`if`, `if/else`, and `switch` statements) and iteration statements (`while`, `do/while`, and `for` statements) and their associated substatements are considered to be blocks. [6.8.4, Selection statements, and 6.8.5, Iteration statements]

Example

```
int i = 42;
for (int i=5, j=15, i<10; i++, j--) // i and j are accessible
                                    // only within this loop
                                    // statement
   printf ("loop %i %i\n", i, j); // prints the values of i and
                                  // j defined within the loop
printf ("i = %i\n", i); // prints i = 42
```

intmax for Preprocessor Expressions

Preprocessor expressions are evaluated using `intmax_t` and `uintmax_t`. `intmax_t` and `uintmax_t` are defined in `<stdint.h>` to be long long and unsigned long long, respectively. Previously, `int` and `unsigned int`, respectively, were used. [6.10.1, Conditional inclusion]

Variadic Macros

Variadic function-like macros are allowed. A variadic macro is a macro that can be declared to have a variable number of arguments. [6.10.3, Macro replacement]

Example

```
#define debug (...)   fprintf (stderr, __VA_ARGS__)
// The ellipsis (...) matches an arbitrary positive number of
// macro arguments that can be referred to by __VAR_ARGS__
// includes the separating commas)
debug ("flag");
// expands to:  fprintf (stderr "flag");
debug ("x = %i\n", x);
// expands to:  fprintf (stderr "x = %i\n", x);
```

c99 Full Support

The TNS, TNS/R, TNS/E, and TNS/X Compilers all conform to the 1989 ANSI C Standard. The C Standard itself was updated in 1999 and is commonly referred to as “c99”. For H06.08 and later H-series RVUs and J06.03 and later J-series RVUs, the TNS/E C Compiler and TNS/E C Standard library and for L-series RVUs, the TNS/X C Compiler and TNS/X C Standard library support a selected subset of new features that were added by the 1999 standard. This subset is available when you use the C99LITE pragma and are described in **c99 Selected Features (C99LITE)** on page 598.

For H06.21 and later H-series RVUs, J06.10 and later J-series RVUs, and L-series RVUs, Full support for the c99 standard is available for TNS/E and TNS/X native applications:

- Full support is only provided for programs using IEEE floating point format. Complex types and several new math functions and macros are not supported for Tandem floating point format. However, all other c99 features are available when compiled using the Tandem floating point format option. These features are supported only by the TNS/E and TNS/X native C compilers; they are not supported by the TNS C, TNS/R C, or C++ compilers.
- Support for embedded SQL/MX, but not SQL/MP is available with `c99`.
- The `VERSION1` and `VERSION2` pragmas are not supported.

If your application maintains common source for TNS- or TNS/R-targeted compilations, HPE recommends that you protect the usage of c99 features by conditional compilation within `#if` or `#ifdef` statements. Applications that use any of the new runtime library features (described in Chapter 7 of the standard) will not run on systems running H06.20 or earlier H-series RVUs or J06.09 or earlier J-series RVUs.

Support for the 1989 C Standard and set of features enabled by using the `C99LITE` pragma are still provided for the `c89` compiler utility.

This appendix is not intended to replicate the ISO/IEC 9899:1999 standard. For detailed information about c99 features, see the standard document.

Enabling C99

To enable c99 support:

- When compiling on Guardian, use the C99 option of the CCOMP command to enable compiling to the 1999 standard (for example `CCOMP/ in filec/;c99`. The default is to compile according to the 1989 standard.
- When compiling on OSS and Windows, use `c89` to compile using the 1989 standard or use `c99` to compile using the 1999 standard.

Compiler Utility Differences

Most of the options supported by the `c99` compiler utility are identical to `c89` options. There are a few differences:

- The flag to specify optimization level is different. For `c89`, the flag is: `-Woptimize=`*n*. For `c99`, the flag is: `-O`*n*. For both, *n* specifies the optimization level, and must be one of 0, 1, or 2.
- Support for c99 is available for TNS/E and TNS/X-targeted compilations only, so flags that apply to TNS/R compilations, such as `-Wtarget=mips` and `-Wtarget=tns/r`, are not supported by the `c99` utility.

- None of the `-Wsql` flags for SQL/MP are supported by `c99`.
- The `c99` utility does not support the `-Wc99lite` flag.
- The `c99` utility does not support the `-Wversion2`, `-Wversion1`, or `-Wmigration_check` flags.
- The `c99` utility does not support the `-Wkr` flag.

For more information about the `c99` utility, see the `c99(1)` reference page either online or in the *Open System Services Shell and Utilities Reference Manual*.

Functions that Do Not Support Tandem Floating Point

These functions support the IEEE floating point format but not the Tandem floating point format:

```
exp2()
exp2f()
exp2l()
fdim()
fdimf()
fdiml()
fma()
fmaf()
fmal()
fmax()
fmaxf()
fmaxl()
fmin()
fminf()
fminl()
log2()
log2f()
log2l()
lrint()
lrintf()
lrintl()
llrint()
llrintf()
llrintl()
lround()
lroundf()
lroundl()
llround()
llroundf()
llroundl()
nan()
nanf()
nanl()
nearbyint()
nearbyintf()
nearbyintl()
nexttoward()
nexttowardf()
```

```
nexttowardl()
remquo()
remquof()
remquol()
round()
roundf()
roundl()
scalbn()
scalbnf()
scalbnl()
scalbln()
scalblnf()
scalblnl()
tgamma()
tgammaf()
tgammal()
trunc()
truncf()
truncl()
```

Function-Like Macros that Do Not Support Tandem Floating Point

These function-like macros support the IEEE floating point format but not the Tandem floating point format:

```
fpclassify()
signbit()
isfinite()
isinf()
isnan()
isnormal()
isgreater()
isgreaterequal()
isless()
islessequal()
islessgreater()
isunordered()
```

c11 support

Starting from L16.05 RVU, the TNS/X Compilers conform to the 2011 ANSI C Standard. The C Standard was updated in 2011 and is referred to as `c11`. `c11` supports compilation of C and C++ programs. The `c11` standards are not compatible with previous standards. Not all `C89` and `C99` programs are valid as a `c11` program. You must explicitly choose to compile to the `c11` standard.

Starting from L17.02, `c11` extends its support to these C++ options:

- `—WBoost`
- `—Wbuild_neutral_library`
- `—Wcplusplus`
- `—Wexception_safe_setjmp`
- `—Wexport_typeid`
- `—Wforce_static_typeinfo`
- `—Wforce_static_vtbl`
- `—Winline_compiler_generated_functions`
- `—Winline_limit`
- `—Wnoexceptions`

Starting from L17.08 RVU, C++11 threads are supported with the `-Wthread` option. The `-Wthread` option is only supported with `-Wsystype=oss`.

Support on Guardian platform

Use the `c11` option for `CCOMP` command to compile `c11` standard programs.

Support on Windows and OSS platforms

Use the `c11` utility to compile `c11` standard programs. The options supported by `c11` utility are same as `c99` utility with the following exceptions:

- `c11` does not support `tns/e` or `ipf` arguments for `—Wtarget`.
- `c11` does not support `H06.nn` or does not support `J06.nn` as arguments to `-WRVU`.
- `c11` does not support the `-Weld` or `-Weld_obey` options.

Compiler options

The `c11` specific compiler options are:

`-WMMD` –directs the compiler to write makefile dependency information for the source file being compiled to a file with the same base name, but with a `.md` filename extension. System header files are omitted.

Files in the current working directory are listed using relative path names. All other files are listed with absolute path names.

`—Wpreinclude=`*`filename`* –directs the compiler to process the *filename* file as if `if #include` *`filename`* is included as the first line of the primary source file.

c++11 Support

Starting with L17.02 RVU, the TNS/X compilers conform to the 2011 ANSI C++ Standard. The C++ Standard was updated in 2011 and is referred to as `c++11`. The `c++11` standard is not compatible with previous C++ standards. Not all `c++98` programs are valid as a `c++11` program. User must explicitly choose to compile `c++11` standard.

Both tandem floating point and IEEE floating point are supported. But, floating point concepts such as NaN and infinity are only supported for IEEE floating point.

Support on Windows and OSS platforms

Use the `c11` utility to compile `c++11` standard programs.

Support on Guardian platform

Use `VERSION4` option of the `CPPCOMP` command to enable compiling to the 2011 standard (for example `CPPCOMP/ in filec/;VERSION4)`. The default standard is `VERSION3`. The following options are not supported when `VERSION4` option is specified:

- TARGET TNS/E
- RVU H06.nn
- RVU J06.nn